UNIVERSIDADE FEDERAL DE SERGIPE
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO

DIEGO ASSIS SIQUEIRA GOIS

JOÃO PAULO ANDRADE LIMA

SIMULAÇÃO DE PILHAS DE PROTOCOLOS PARA INTERNET DAS COISAS

São Cristovão
2014

DIEGO ASSIS SIQUEIRA GOIS
JOÃO PAULO ANDRADE LIMA

SIMULAÇÃO DE PILHAS DE PROTOCOLOS PARA INTERNET DAS COISAS

Trabalho de conclusão de curso apresentada ao Curso de Engenharia de Computação da Universidade Federal de Sergipe como requisito parcial para a obtenção do título de bacharel em Engenharia de Computação.

ORIENTADOR: Prof. Dr. Admilson de Ribamar Lima Ribeiro

São Cristovão
2014

DIEGO ASSIS SIQUEIRA GOIS
JOÃO PAULO ANDRADE LIMA

SIMULAÇÃO DE PILHAS DE PROTOCOLOS PARA INTERNET DAS COISAS

Trabalho de conclusão de curso apresentada ao Curso de Engenharia de Computação da Universidade Federal de Sergipe como requisito parcial para a obtenção do título de bacharel em Engenharia de Computação.

Aprovada em ____/____/_____
BANCA EXAMINADORA

_____
Prof. Dr. Admilson de Ribamar Lima Ribeiro
Universidade Federal de Sergipe

_____
Prof. Dr. Marco Túlio Chella
Universidade Federal de Sergipe

_____
Prof. Dr. Ricardo José Paiva de Britto Salgueiro
Universidade Federal de Sergipe

Gois, Diego Assis Siqueira
Simulação de pilhas de protocolos para internet das coisas / Diego Assis Siqueira Gois, João Paulo Andrade Lima; orientador: Dr. Admilson de Ribamar Lima Ribeiro – São Cristovão: 2014.

DEDICAMOS

A Deus, autor e consumador da fé.
Aos nossos pais, pelo apoio, colaboração e confiança.

AGRADECIMENTOS

Agradeço em primeiro lugar a Deus, grande arquiteto do universo, por ter me guiado nesta longa estrada, iluminando meus passos para que meu objetivo fosse atingido com êxito.

Ao meu pai José Assis (*in memorian*) e minha mãe Maria Edivalda que não mediram esforços e me apoiaram em todos os aspectos para que eu chegasse até este momento com carinho, amor e dedicação.

A minha irmã e segunda mãe Sheilla por todo incentivo, carinho, amor e cumplicidade, à minha sobrinha e afilhada Natália que esteve comigo em todos os momentos desde seu nascimento e que tenho grande amor.

A UFS (Universidade Federal de Sergipe), seu corpo docente, direção e administração que oportunizaram a janela que hoje vislumbro um horizonte superior, eivado pela acendrada confiança no mérito e ética aqui presente.

Ao colega e amigo João Paulo, o qual tive o prazer de dividir conhecimento e momentos durante todo o ensino acadêmico, inclusive neste trabalho.

Ao colega Marcus Vinicius que ajudou em alguns pontos importantes deste trabalho.

Ao meu orientador, professor Dr. Admilson de Ribamar, pelas oportunidades fornecidas além do incentivo e cobranças necessárias.

E a todos que direta ou indiretamente fizeram parte da minha formação, o meu muito obrigado.

Diego Assis Siqueira Gois

AGRADECIMENTOS

Gostaria de agradecer a Deus e todas as suas variações, ao universo por me oferecer caminhos que permitiram este momento.

Aos meus pais Neide e Hosanito (Binha) por estarem sempre ao meu lado, pelo carinho e amor. Em especial a minha mãe que sempre me incentivou e buscou oferecer caminhos para uma boa educação.

A minha irmã Maria Loranny pelo carinho, afeto e apoio.

Ao meu amor Mariana Lima, por todos os conselhos, ajudas, apoio, compreensão, incentivos, carinhos e companheirismo. E por sempre estar ao meu lado me fortalecendo e me completando.

Ao amigo Diego Assis, por todo esforço realizado neste trabalho e por todas as lutas conquistadas em parceria.

Ao orientador Prof. Dr. Admilson de Ribamar, pelas oportunidades e orientação em todas as monitorias. Agradeço por ser tão presente e participativo na monografia.

Aos professores doutores da banca examinadora Marco Túlio e Ricardo Salgueiro pela participação na avaliação deste documento.

A todos os professores que ajudaram de forma direta ou indireta na construção de uma base didática para produção desse trabalho.

João Paulo Andrade Lima

“Obstáculos e dificuldades fazem parte da vida. E a vida é a arte de superá-los”.

Luis Sérgio Álvares DeRose

RESUMO

A Internet das Coisas (*IoT*) se caracteriza por objetos do dia-a-dia que interagem entre si e com o ambiente ao redor, formando uma rede para oferecer comodidade e praticidade para seu usuário. Analisando o cenário crescente acerca da Internet das Coisas, o fato de ser uma área em expansão, recente, pouco estudada e documentada; e da necessidade de se ter resultados para servirem de base para estudo e trabalhos futuros. Este trabalho de Conclusão de Curso (TCC) intitulado *Simulação de pilhas de protocolos para internet das coisas*, visa produzir um manual, que mostra a simulação dos protocolos UDP, *simple* UDP, TCP, uma aplicação cliente-servidor REST sobre a pilha de comunicação *uIP* e o módulo *Anonymous best-effort local area broadcast* (abc) sobre a pilha de comunicação Rime. Este trabalho utiliza a ferramenta COOJA, que está incluído no sistema Contiki, para a realização das simulações e padroniza nestas o uso da versão seis do protocolo IP. Como resultado, este documento serve de fomento de estudo para trabalhos e pesquisas futuras na área em questão.

Palavras-chave: Internet das Coisas. Contiki. COOJA. Simulação.

ABSTRACT

The Internet of Things (*IoT*) is characterized by day-to-day objects that interact with each other and with the surrounding environment, forming a network to provide comfort and convenience to its user. Analyzing growing scenario about the Internet of Things, the fact of being an expanding area, latest, little studied and documented; and the necessity of have results as a basis for study and future work. This work of Course Completion (TCC) entitled *Simulation protocol stacks for internet of things*, aims to produce a manual that shows the simulation of the UDP protocols, simple UDP, TCP, a REST client-server application on the communication stack *uIP* and the best Anonymous -effort local area broadcast (abc) module on the Rime communication stack. This work uses the COOJA tool, which is included in Contiki system for carrying out these simulations and standardizes the use of version six of the IP protocol. As a result, this document serves to feed study for future researches and works in the area in question.

Keywords: Internet of Things. Contiki. COOJA. Simulation.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS, SIGLAS E SÍMBOLOS

ABC	Anonymous Best Effort Broadcast
ADC	Analog-to-Digital Converter
API	Application Programming Interface
CPU	Central Processing Unit
dBM	Decibel Miliwatt
DCA	Digital-to-Analog Converter
DNS	Domain Name System
FDM	Frequency Division Multiplexing
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IoT	Internet of Things
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IRQ	Interrupt Request
JNI	Java Native Interface
LED	Light-Emitting Diode
LPM	Landau-Pomeranchuk-Migdal
Mhz	Megahertz
MIT	Massachusetts Institute of Technology
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit
NMAP	Network Mapper
RAM	Random Access Memory
REST	Representational State Transfer
RFID	Radio-Frequency Identification
RSSI	Received Signal Strength Indication
RPL	IPv6 Routing Protocol for Low Power and Lossy Networks
RX	Taxa de Sucesso de Recepção
SNMP	Simple Network Management Protocol

TCP	Transmission Control Protocol
TDM	Time Division Multiplexing
TX	Taxa de Sucesso de Transmissão
UDP	User Datagram Protocol
USB	Universal Serial Bus
COAP	Constrained Application Protocol

SUMÁRIO

1 INTRODUÇÃO

Não deveria existir uma rede que fizesse todos os nossos dispositivos colaborarem em todos os momentos, conversar espontaneamente entre si e com o resto do mundo, e todos juntos comporem uma espécie de computador virtual único – a soma de sua inteligência e conhecimento? (HALADJIAN, 2005). Esse avanço tecnológico no cenário *wireless* alcançou o mundo físico com a integração de sensores de redes e a incorporação de tecnologia de comunicação em objetos da vida cotidiana. Essa ideia de ascensão que a Internet da época teria, foi inventado pelo fundador do *MIT Auto-ID Center*, Kevin Ashton em 1992 (ASHTON, 2009) e foi chamado de Internet das Coisas (*Internet of Things - IoT*). Considerando termos históricos, Kevin Ahston previu a mudança de processamento de informação do computador para o sensoriamento do computador (SANTUCCI, 2010).

Contudo, percebe-se pela maioria dos artigos sobre a *IoT*, que a sua definição ainda é bastante confusa e sujeita a debate filosófico (UCKELMANN et al, 2009). Porém, uma ideia básica da Internet das Coisas, é a presença generalizada em torno de nós de uma variedade de coisas ou objetos - tais como marcadores de identificação por radiofrequência (*Radio-Frequency Identification - RFID*), sensores, atuadores, telefones celulares, etc, que através de esquemas de endereçamento único, são capazes de interagir um com o outro e cooperar com seus vizinhos alcançando objetivos comuns (GIUSTO et al, 2010).

Essa interoperabilidade obtida pela interação dos objetos influencia tanto o ambiente familiar quanto os campos de trabalhos, surgindo cenários como casas e escritórios inteligentes, aprendizagem avançada, vida assistida, cuidados de saúde (*e-health*), entre outros. Além de proporcionar a eles um maior grau de inteligência, permitindo a sua adaptação e comportamento autônomo, enquanto garante confiança, privacidade e segurança (ATZORI et al, 2010) .

De outro lado, a *IoT* insere vários novos problemas, dentre os quais um muito relevante é sobre os aspectos de rede, pois com a variedade de tecnologias existentes, pode tornar complexa a comunicação destes objetos através da mesma. Para superar este obstáculo, os *softwares* de comunicação de rede utilizam-se de *middleware* para ocultar os detalhes de diferentes tecnologias e dispensar o programador de questões que não são relevantes para o desenvolvimento de sua aplicação, fato este, que é possibilitado pelas infraestruturas da *IoT*. O *middleware* está ganhando cada vez mais importância nos últimos anos devido ao seu importante papel na simplificação do desenvolvimento de novos serviços e a integração de tecnologias herdadas dentro de outras novas (ATZORI et al, 2010).

Além disso, outro problema referente à comunicação entre objetos é a limitada capacidade de armazenamento, processamento e energia que tais objetos possuem. Como uma possível solução para o problema de capacidade limitada, Adam Dunkels (2003) apresentou uma implementação da pilha de comunicação TCP/IP (conhecida assim devido à presença dos protocolos: *Transmission Control Protocol* e *Internet Protocol*) para dispositivos integrados com baixíssimo poder de processamento, mais especificamente, para plataformas de oito e dezesseis bits. Para obter tal feito, a implementação do *uIP* (como ficou conhecida tal pilha) é projetada para ter apenas o conjunto mínimo de recursos necessários para o funcionamento da pilha TCP/IP.

Outro fato preocupante da Internet das Coisas é o fato da identificação única dos objetos, pois cada um desses deve ser recuperável por qualquer usuário independente da sua posição. E com a inserção de um grande número de objetos na rede e consequentemente, um número extremamente elevado de nós, é exigido uma grande quantidade de endereços que devem ser exclusivos.

Porém, atualmente o protocolo IPv4 identifica cada nó através de um endereço de 4 bytes e sabe-se que o número de endereços IPv4 disponíveis está diminuindo rapidamente e em breve se esgotará. Neste caso, é aconselhável utilizar o protocolo IPv6, pois endereços IPv6 são expressos por meio de 128 bits e, portanto, é possível definir 10^{38} endereços, que devem ser suficientes para identificar qualquer objeto que vale a pena de ser incluído na rede (ATZORI et al, 2010).

Analisando todo este cenário crescente acerca da Internet das Coisas e conhecendo o trabalho (DUNKELS et al, 2004) que apresenta o sistema operacional Contiki e o conhecimento sobre o simulador COOJA, incluído neste, que permite a simulação simultânea no nível de rede, no nível do sistema operacional, e no código de nível de conjunto de instruções da máquina; além do trabalho de (PETERSON et al, 2003) onde os autores realizam várias simulações de redes de computadores utilizando um simulador em especial, e publicam os resultados na academia para estudo.

1.1. Objetivo

O objetivo deste trabalho é simular as pilhas de comunicação *uIP* e Rime utilizando o protocolo IPv6 no contexto da *IoT* utilizando o simulador COOJA, e publicar os resultados na academia objetivando fornecer base didática para o estudo e a pesquisa futura na área em questão.

1.2. Metodologia

Inicialmente realizou-se uma revisão bibliográfica a respeito da Internet das Coisas para obter embasamento teórico do tema abordado. Posteriormente efetivou-se a pesquisa de temas relacionados, como a pilha de comunicação *uIP*, pilha de comunicação Rime, protocolos IPv4 e IPv6.

Após o levantamento bibliográfico, ocorreu uma busca pelas tecnologias que seriam utilizadas para a simulação das pilhas e optou-se pelo sistema operacional Contiki (DUNKELS, 2004) devido a suas características e pela presença do simulador COOJA, o qual vai servir como ferramenta fundamental do trabalho.

Em seguida foi feito um profundo estudo da documentação do sistema operacional Contiki junto com o simulador COOJA, para então dar início as simulações que são descritas neste trabalho. Com as simulações prontas, os resultados começaram a ser analisados para geração de gráficos e tabelas.

Por fim, foram observados resultados dentro do esperado e sugerimos algumas possíveis alterações, complementos ou execução em plataforma real dos nossos códigos para trabalhos futuros.

1.3. Organização do Trabalho

Este trabalho foi divido em vários capítulos, sendo o primeiro capítulo a introdução. O capítulo seguinte trata-se da fundamentação teórica, onde são expostos conceitos necessários para o entendimento do trabalho. O capítulo três expõe e detalham as ferramentas utilizadas na execução do trabalho, neste caso, o sistema operacional Contiki, o simulador COOJA e a ferramenta de desenvolvimento Make. O capítulo quatro retrata os trabalhos relacionados ao tema apresentado como objetivo central deste trabalho.

O capítulo cinco trata das simulações em *IoT*, no qual é apresentado a montagem do ambiente, o detalhamento da implementação de cada simulação e os resultados obtidos nos mesmos. Já o capítulo seis mostra as considerações finais sobre o tema abordado e as propostas de trabalhos futuros. Além do anexo onde estão presentes todos os códigos utilizados neste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

O objetivo deste capítulo é apresentar conceitos que fundamentam o tema abordado, e que além de servirem como base para o entendimento da *IoT* de modo geral são partes fundamentais do mesmo. Nas seções a seguir são abordados os seguintes temas respectivamente: apresentação do sistema, pilha de comunicação *uIP*, pilha de comunicação Rime, os protocolos IPv4 e IPv6, TCP, UDP, *Tmote Sky*, REST.

2.1 Apresentação do Sistema

A *IoT* consiste em um conjunto de objetos inteligentes com conexão a Internet e endereçamento único, o que permite comunicação entre eles e servidores para diversos fins, os quais são possíveis destacar o uso doméstico para trazer comodidade, lazer e saúde, ou o uso comercial através de automação, logística, gerenciamento de negócios e o transporte inteligente de pessoas e mercadorias. Estes objetos utilizam tecnologias como *RFID*, sensores, atuadores e rede *wireless*.

A *IoT* pode residir em diversas coisas do dia a dia como embalagens de produtos alimentícios, mobília e documentos. Porém a sua difusão traz algumas ameaças, visto que à medida que os objetos do dia a dia apresentam riscos de segurança de informação, a *IoT* contribuirá para o crescimento destes.

Contudo para que essa difusão seja efetivada, alguns desafios precisam ser vencidos, como o uso do IPV6 em substituição ao IPV4. Isso ocorre devido ao grande número de endereços necessários para o endereçamento único dos objetos inteligentes, obtenção de interoperabilidade completa entre os dispositivos interconectados, garantir confiança, privacidade e segurança, prover alto grau de inteligência (de forma a possibilitar seu comportamento autônomo), superar alguns problemas em relação aos aspectos de rede (quantidade, tamanho, tempo de vida e velocidade que os dados são transportados) e prover de forma eficiente energia para milhares de objetos.

A arquitetura mais comum para *IoT* é mostrada na figura 1, onde é possível notar vários objetos inteligentes reunindo informações e as enviando pela Internet para que o usuário possa retirar o maior benefício possível das mesmas.

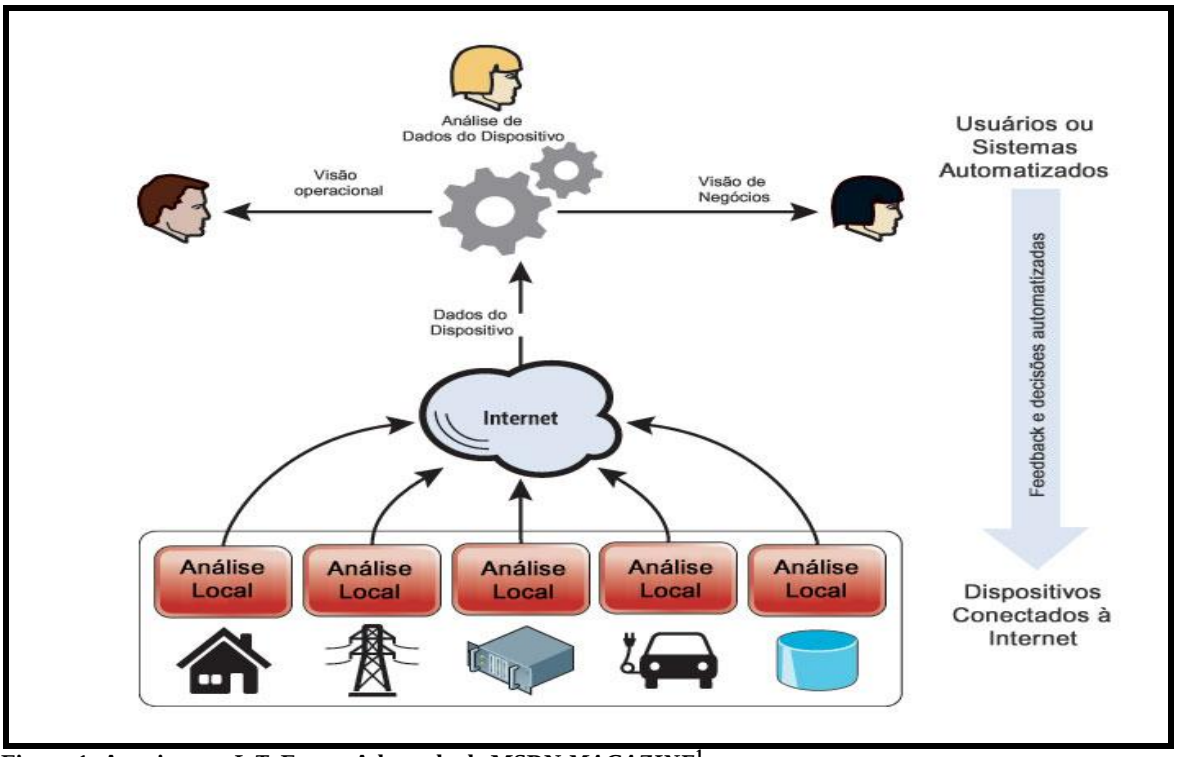

Figura 1: Arquitetura *IoT*. Fonte: Adaptado de MSDN *MAGAZINE*[1]

Para que os dispositivos enviem informações via Internet é necessário que todos eles se comuniquem via IP. Porém a pilha TCP/IP é pesada para ser implantada em pequenos dispositivos, por isso, uma pilha mais leve, porém com total interoperabilidade com IP, chamada *uIP* foi desenvolvida para ser utilizada por esses dispositivos. A figura 2 mostra o caminho feito por um pacote na entrada ou saída de uma pilha *uIP*.

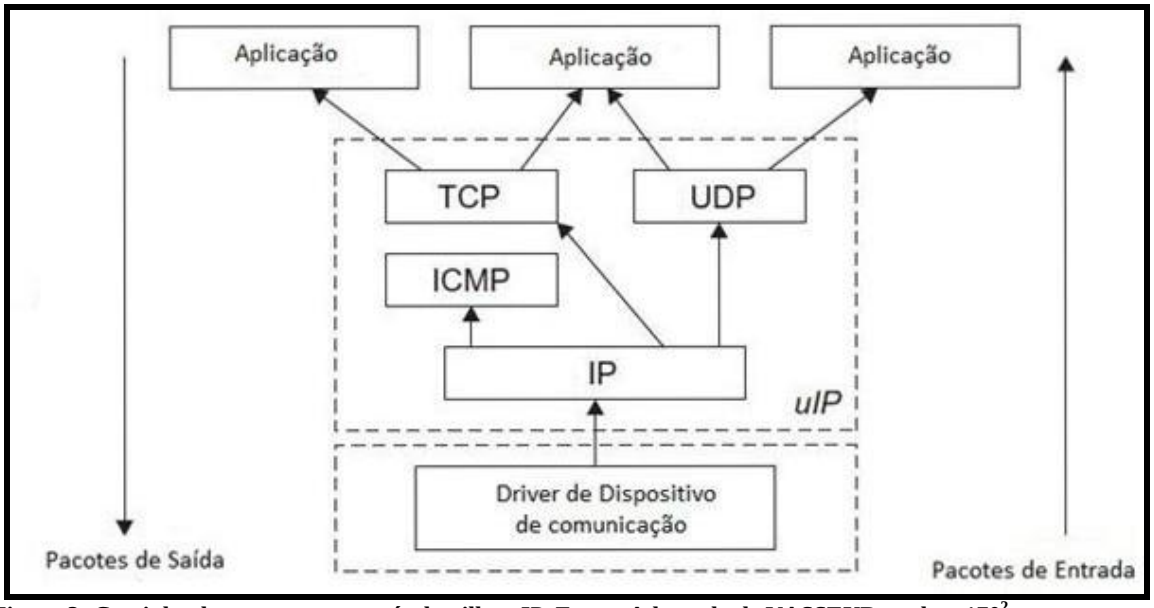

Figura 2: Caminho de um pacote através da pilha *uIP*. Fonte: Adaptado de VASSEUR et al, p. 170[2].

[1] Disponível em: http://msdn.microsoft.com/pt-br/magazine/hh852591.aspx;. Acesso em maio de 2014.
[2] VASSEUR, J.P.; DUNKELS, A. **Interconnecting Smart Objects With IP.** The Next Internet. Morgan Kaufhann.

2.2 Pilha de Comunicação *uIP*

O trabalho em questão é baseado em torno da inserção dos objetos de uso do dia a dia na Internet pra assim obter benefícios e facilidades em serviços prestados pelos mesmos. Essa facilidade se dá pela interação dos objetos através de sensores *wireless* como identificadores de radiofrequência, porém para esse contato ocorrer, é necessária uma troca de mensagens através da rede, esse trabalho é realizado atualmente pela pilha TCP/IP (POSTEL, 1981).

Contudo, o principal problema para a integração reside na natureza restrita dos dispositivos embarcados. Limitações típicas são: fonte de alimentação limitada, tamanho de memória limitado e baixo poder de processamento. Outro problema é o grande número de tipos de dispositivos, enquanto que cada tipo tem diferentes características, limitações e habilidades de comunicação física (KLAUCK et al, 2012). Devido a estas limitações, a pilha TCP/IP tradicional não pode ser utilizada nestes dispositivos, ferindo assim o meio de troca de mensagens e consequentemente, a interação entre os mesmos.

Sabendo que as implementações tradicionais TCP/IP têm exigido muitos recursos, tanto em termos de tamanho do código e utilização da memória para ser útil em sistemas de oito ou dezesseis bits, estes requisitos tornaram impossíveis o uso da pilha TCP tradicional em sistemas com pouca memória RAM e espaço para código (DUNKELS A., 2003). Ainda assim, objetivando utilizar a pilha TCP/IP em tais sistemas e conhecendo as limitações destes sistemas, Adam Dunkels implementou uma pilha TCP/IP portátil e genericamente pequena, e chamou-a de *uIP* (DUNKELS, 2003).

Para atingir tal feito DUNKELS (2003) projetou o *uIP* com apenas o conjunto mínimo de recursos necessários de uma pilha TCP / IP. Como resultado dessa alteração, o *uIP* somente lida com uma única interface de rede e não implementa o protocolo UDP, porém possui como foco central os protocolos IP, ICMP e TCP.

Deste modo, devido ser mais leve e a sua adaptabilidade aos sistemas com baixos recursos (pouco processamento e pouca memória), a pilha de comunicação *uIP* acabou se encaixando no âmbito da Internet das Coisas e servindo como um dos pilares principais para a sua execução na prática.

2.3 Pilha de Comunicação Rime

Segundo Dunkels em (DUNKELS, 2007), os primeiros trabalhos em redes de sensores descobriram que arquiteturas de comunicação em camadas tradicionais eram bastante restritivas para redes de sensores. Além disso, as otimizações de camada-cruzada para agregação de dados são muito eficientes, porém conduzia para sistemas frágeis e incontroláveis. Devido a este ponto negativo, foi criada a pilha de comunicação em camadas Rime (DUNKELS, 2007), que simplifica a implementação e a complexidade dos protocolos de comunicação.

Diferentemente das arquiteturas de comunicação tradicionais, as camadas de abstração da pilha Rime são extremamente simples tanto em relação à interface quanto a implementação e são combinadas para formar abstrações poderosas em alto nível. Por ser tão simples e genérica, cada camada pode ser utilizada para objetivos diferentes. Desse modo, a pilha acaba por simplificar a implementação de protocolos de redes de sensores e facilitando o reuso de código.

Rime possui uma arquitetura limitada que não permite uma estrutura totalmente modular, podendo somente ser substituída a camada mais baixa, que é o módulo abc (*anonymous best effort broadcast* – permite uma abstração de um canal de dezesseis bits em que envia um broadcast sem endereçamento) e a camada de aplicação.

Porém, a presença do código de Rime é pequena e leve não ultrapassando dois *quilobytes* e exigindo requisitos de memória por volta de dez *bytes*.

2.4 Protocolo IPv4 e IPv6

Com a possibilidade de ser executado no mundo real, a *IoT* enfrenta outra questão na conjectura atual, a identificação exclusiva dos objetos inseridos na rede. De fato, essa identificação ocorre através do protocolo de Internet (IP), pois o protocolo IPv4 (o qual está em vigor atualmente) assume que o endereço IP de um nó identifica exclusivamente o ponto de ligação do nó à Internet. Portanto, um nó deve ser localizado na rede indicada pelo seu endereço IP, a fim de receber os datagramas destinados a ele, caso contrário, datagramas destinados ao nó não seriam entregues (POSTEL, 1981).

Porém, como foi dito na introdução, o IPv4 se caracteriza por possuir um endereço de 32 bits o que possibilita 2^{32} (4.294.967.296) diferentes endereços. Este endereço é totalmente abstrato e define um endereço único para cada host ligado numa rede

heterogênea independente do hardware e do sistema operacional utilizado (FILHO, 2003). Além disso, cada endereço é dividido em duas partes específicas, na primeira parte identifica-se a rede em que o host está localizado, e na segunda parte, identifica-se o host na rede.

Contudo, como hoje a Internet tornou-se a maior rede de dados do mundo e vêm crescendo cada vez mais, os 4.294.967.296 diferentes endereços oferecidos pelo IPv4 estão se esgotando, e infelizmente não suportará a inserção de dezenas de milhares de objetos, que é a proposta da Internet das Coisas.

A fim de resolver os problemas existentes no IPv4, DEERING (1998) projetou o IPv6. O IPv6 aumentou o tamanho do endereço IP de 32 bits para 128 bits, o que possibilita 2^{128} diferentes endereços, configurando um enorme espaço de endereçamento (milhares de bilhões), o que acaba tornando-o um número difícil de ser apresentado. Além de melhorar a escalabilidade do roteamento *multicast* através da adição de um campo chamado "escopo" para endereços *multicast*, ele passou a oferecer também outros serviços como conectividade segura e maior qualidade de serviços.

Com o surgimento do IPv6 e consequentemente a criação de milhares de bilhões de novos endereços exclusivos, o impasse na execução de por em prática a *IoT* diminui, pois haverá endereços disponíveis para as dezenas de milhares de objetos que podem ser inseridos na rede. Essas alterações no cenário mundial, e suas respectivas soluções, acabam comprovando a necessidade do uso da pilha uIP com o protocolo IPv6.

2.5 TCP

O TCP (*Transmission Control Protocol*) é o protocolo de transporte confiável da camada de transporte, orientado a conexão, da Internet (KUROSE et al, 2010). Este protocolo usa a conexão ponto a ponto, ou seja, existe apenas um remetente e um destinatário, sendo comum a comunicação entre um cliente e um servidor. A propriedade ponto a ponto impede que sejam realizadas transferências de mensagens de um remetente para vários destinatários, o chamado *multicast*, comum em conexões UDP.

O mesmo é dito orientado a conexão, pois antes que um remetente comece a enviar segmentos para um destinatário é necessário uma comunicação entre os mesmos, nesta comunicação inicial são trocados segmentos a fim de estabelecer parâmetros na transferência de dados e iniciar as variáveis de estado do TCP. A abertura da conexão é feita através desta troca de segmentos, onde um cliente envia um segmento de configuração sem mensagens ao

servidor, que por sua vez responde com outro segmento de configuração e por fim o cliente envia um novo segmento (podendo conter mensagens) ao servidor, finalizando o procedimento chamado de *handshake*.

A conexão TCP não é um circuito TDM (*Time Division Multiplexing*) ou FDM (*Frequency Division Multiplexing*) fim a fim, como acontece em uma rede de comutação de circuitos. Tampouco é um circuito virtual, pois o estado de conexão reside inteiramente nos dois sistemas finais (KUROSE et al). Por isso, elementos intermediários de rede não mantém o estado da conexão, estes veem apenas segmentos e não qualquer tipo de conexão.

O serviço oferecido pelo TCP é do tipo *full-duplex*, isto é, dados vindos da aplicação podem viajar pela conexão em ambos os sentidos, assim é possível que um dado do cliente viaje para o servidor ao mesmo tempo em que o dado do servidor viaja para o cliente. Enquanto a conexão esta ativa, o TCP utiliza suas características principais, tais como: entrega ordenada dos dados, controle de fluxo, controle de congestionamento, detecção de falhas, temporizadores (que ajustam o controle do atraso na rede), janela deslizante e dados urgentes para garantir uma entrega confiável dos dados e um melhor desempenho.

O tamanho do segmento TCP para envio é limitado pelo MSS (*Maximum Segment Size*), que por sua vez utiliza o MTU (*Maximum Transmission Unit*) como limitação, o MTU pode ser visto como o maior tamanho possível de um segmento na camada de enlace.

Implementações tradicionais de TCP / IP têm exigido muitos recursos tanto em termos de tamanho de código e uso de memória para ser útil em sistemas de oito ou dezesseis bits (CONTIKI, 2012). Assim para tais sistemas utiliza-se a pilha *uIP* que implementa apenas o conjunto mínimo de recursos necessários ao funcionamento do TCP, o que permite retirar alguns dos mecanismos como: janela deslizante, controle de fluxo, controle de congestionamento, dados urgente e cálculos de verificação em prol de uma considerável economia de memória aos sistemas de pequeno porte.

Apesar da maioria das implementações TCP utilizarem a janela deslizante para enviar dados em sucessão sem que seja necessário esperar uma confirmação para cada segmento, o *uIP* não guarda pacotes enviados e permite que somente um segmento TCP por conexão receba uma confirmação de entrega (frequentemente chamado de *acknowledgmen*t ou *ack*) em determinado tempo, inviabilizando a implementação deste mecanismo, já que tornará a aplicação mais complexa por não guardar os pacotes enviados.

O controle de fluxo é implementado no TCP para prover comunicação entre os mais variados tamanhos de memória de diferentes *hosts*, onde o destinatário da mensagem indica em uma janela o tamanho disponível em seu buffer. Em *uIP*, a aplicação não pode

enviar mais dados do que o *host* receptor possa armazenar, a aplicação também não pode enviar mais dados do que a quantidade de bytes enviadas permitida pelo servidor (CONTIKI, 2012).

Como *uIP* manuseia apenas um segmento TCP por conexão e o controle de congestionamento serve justamente para limitar o número de conexões simultâneas TCP na rede, este mecanismo não se faz necessário.

Na maioria das implementações TCP, o mecanismo de dados urgentes é adicionado através de uma API assíncrona. Como o *uIP* utiliza uma API baseada em eventos assíncronos, dados urgentes não conduz um aumento de complexidade.

2.6 UDP

A camada de transporte da pilha *uIP* provê uma comunicação lógica entre processos de aplicação, deste modo, para a aplicação é como se os nós estivessem diretamente ligados. O protocolo UDP (*User Datagram Protocol*) aparece como uma das opções de comunicação nesta camada, este fornece um serviço não confiável, pois não há garantias que os pacotes (também chamados de datagramas) serão entregues ao destinatário. Outra característica marcante é o fato de não ser orientado a conexão, ou seja, não é necessária a abertura de uma conexão entre os nós antes que os mesmos possam se comunicar.

O protocolo UDP, um protocolo de transporte extremamente simples, implementa o mínimo necessário na camada de transporte, esse protocolo pega as mensagens vindas da camada de aplicação, adiciona quatro campos de cabeçalho e envia o segmento para camada de rede. No cabeçalho são encontrados as portas do *host* fonte e o destinatário da mensagem para que a mensagem seja demultiplexada corretamente, caso o segmento chegue ao destino, o comprimento do total do segmento, a soma de verificação (usada para detectar erros) e a mensagem vinda da aplicação.

O uso do UDP se torna viável em algumas situações em que não é necessária uma transferência confiável de dados, como por exemplo, em aplicações de tempo real (na maioria dos casos são multimídia), estas requerem uma taxa mínima de envio sem grande atraso de transmissão e toleram perda de alguns dados. Neste caso, o controle de congestionamento implementado pelo TCP se comporta muito mal. Entretanto, o TCP está sendo utilizado cada vez mais para transportar mídia, (SRIPANIDKULCHAI et al, 2004) descobriu que aproximadamente 75% do fluxo em tempo real e gravado, utilizaram TCP. Quando as taxas

de perda de pacote são baixas, junto com algumas empresas que bloqueiam o tráfego UDP por razões de segurança, o TCP se torna um protocolo cada vez mais atrativo para transporte de mídia (KUROSE et al, 2010).

Como o UDP apenas envia mensagens sem nenhum estabelecimento de conexão, não há atraso na conexão, tornando bastante interessante o uso do DNS (*Domain Name System*) e do SNMP (*Simple Network Management Protocol*), protocolo de gerenciamento de redes, sobre o UDP.

Por ser difícil de usar a API UDP do Contiki, o mesmo provê uma API mais simples chamada de *simple* UDP que fornece três funções básicas necessárias para uma conexão UDP: registro da conexão, envio de pacote para o endereço IP que a conexão foi registrada e envio de pacote para um IP específico.

2.7 REST

A arquitetura de transferência de estado representacional (*Representational State Transfer*) ou arquitetura REST, é um estilo híbrido derivado de vários estilos arquitetônicos baseados em rede e combinados com restrições adicionais que definem uma interface de conector uniforme (FIELDING, 2000).

Esta arquitetura preza por uma conexão cliente-servidor sem armazenamento de estado, forçando cada requisição do cliente possuir toda a informação necessária para que o servidor retorne os dados desejados. Porém, esta restrição força que o cliente efetue várias requisições ao servidor diminuindo a eficiência da rede. Para resolver isto, REST permite clientes *cache*, que utilizam respostas armazenadas nestes, para solicitações de clientes semelhantes.

Outra característica dessa arquitetura é a utilização de uma interface uniforme, a qual permite uma maior liberdade de implementação restringindo a forma de transferir a informação, o que acaba valorizando a transferência de grande volumes de dados em hipermídia, em detrimento de outras formas de transmissão.

Só que para todas essas restrições darem certo, segundo FIELDING (2000), a abstração fundamental da informação em REST chama-se recurso. Um recurso específico ou seu identificador, em um determinado momento representa um valor ou um conjunto de entidades. E o conector REST, fornece uma interface na qual o cliente pode alterar e acessar o recurso independentemente do tipo de software que está solicitando e do tipo de função.

2.8 Dispositivos Embarcados

Dispositivos embarcados consistem em pequenos circuitos que são microprocessados e possuem tarefas específicas predefinidas anteriormente, sendo o comportamento do dispositivo dependente do modo que o mesmo será usado, por exemplo, o comportamento de um dispositivo embarcado projetado para monitorar um contêiner refrigerador é diferente de outro projetado para monitorar estacionamentos de carros.

A maioria destes dispositivos é equipada com sensores ou atuadores, um pequeno microprocessador, um módulo de comunicação e uma fonte de energia. Tudo isso em poucos centímetros cúbicos, pois os mesmos devem ser pequenos para que possam ser inseridos no dia a dia tanto de indústrias e comércios como na moradia da população.

A interação com o mundo físico através de seus sensores e atuadores e a comunicação (normalmente é utilizada comunicação sem fio) são propriedades fundamentais deste tipo de dispositivos, pois através dos mesmos é possível enviar dados sobre o estado atual do ambiente ou receber dados que ordenam a modificação do ambiente através de seus atuadores.

3 TECNOLOGIAS UTILIZADAS

Neste capitulo serão apresentados conceitos sobre o sistema operacional Contiki e o simulador COOJA, os quais foram utilizados em um ambiente de simulação para Internet das coisas.

3.1. Contiki

As redes de dispositivos sensores sem fio, como por exemplo, RFID, podem conter um grande número destes pequenos dispositivos, os mesmos por possuírem recursos limitados necessitam de um sistema operacional leve o suficiente para permanecer dentro dessas limitações.

O Contiki introduziu a comunicação IP em redes de sensores de baixa potência (CONTIKI, 2012), este é desenvolvido por um grupo de desenvolvedores liderados por Adam Dunkels e contém duas pilhas de comunicação: *uIP* e Rime.

uIP é uma pilha TCP / IP compatível com RFC pequeno que torna possível a comunicação Contiki pela Internet. Rime é uma pilha de comunicação leve projetado para rádios de baixa potência. A mesma fornece uma ampla gama de primitivas de comunicação, de transmissão local de melhor esforço, e um mecanismo de múltiplos saltos de transferência de dados confiável (CONTIKI 2.6, 2012).

O Contiki é um sistema operacional leve com suporte para carregamento dinâmico e substituição de programas individuais (DUNKELS et al, 2004). O mesmo possui código aberto escrito em C e foi transportado para inúmeras arquiteturas, sendo um sistema altamente portátil, flexível e multitarefas e projetado para executar em microcontroladores com baixa quantidade de memória. A flexibilidade tem origem na possibilidade de carregar e descarregar programas individuais dinamicamente nos nós sensores em tempo de execução.

O *kernel* é orientado a eventos, mas o sistema suporta dar preferência à programação *multi-threading* que pode ser aplicado em uma base por processo. De preferência *multi-threading* é implementado como uma biblioteca que está ligada apenas com programas que exigem explicitamente *multi-threading* (DUNKELS et al, 2004).

Este sistema operacional oferece suporte à comunicação IP, neste trabalho utilizamos uma pilha de comunicação *µIP* a qual também é suportada pelo sistema operacional, tanto nas versões quatro (IPv4) como na versão seis (IPv6). *µIP* e Contiki são utilizadas em centenas de empresas em sistemas de navios cargueiros, satélites e

equipamentos de perfuração de petróleo, e são altamente reconhecidos pela popular ferramenta de escaneamento de redes *nmap* (CONTIKI 2.6, 2012).

3.2. COOJA

Desenvolver *software* para redes de sensores pode ser simplificado pelo uso de um sistema simulador, que permita o desenvolvimento de algoritmos, o estudo do comportamento do sistema e a observação das interações em um ambiente controlado (LEVIS et al, 2003). A simulação, uma forma de modelar e prever possíveis resultados em um cenário real pode ser obtida por meio de simuladores.

O COOJA é um simulador flexível baseado em Java e projetado para simular redes de sensores rodando no sistema operacional Contiki (DUNKELS et al, 2004). O mesmo tem a capacidade de simular redes multinível e por isso consegue mesclar a simulação de baixo nível dos sensores de *hardware*, os quais possuem grandes tempos de simulação, com o comportamento de simuladores de alto nível, que fornecem menores tempos de simulação.

Apesar de ser um simulador implementado em Java, o COOJA permite o uso da linguagem de programação C no software de seus nós e a utilização de várias bibliotecas Contiki diferentes carregadas simultaneamente em uma única simulação, isso devido à biblioteca *Java Native Interface* (JNI), que é implementada pelo próprio simulador. O uso do JNI, já citado, oferece ao simulador pleno controle de memória dos nós, o que permite o mesmo visualizar ou alterar as variáveis da simulação no Contiki.

Outra funcionalidade da ferramenta é a opção de simular redes diversificadas, ou seja, executar uma simulação onde cada nó possua diferentes plataformas, por exemplo, uma rede com nós *Tmote sky, Wismote* ou qualquer outra plataforma disponibilizada no simulador, em uma única simulação.

Um nó simulado no COOJA tem três propriedades básicas: sua memória de dados, o tipo de nó e seus periféricos de hardware. Portanto, nós do mesmo tipo executam o mesmo programa sobre o mesmo periférico e são inicializados com a mesma memória de dados. Durante a execução, no entanto, as memórias de dados irão diferir devido a diferentes entradas (ÖSTERLIND et al, 2006).

Os periféricos dos sensores são chamados de interfaces e são responsáveis por avisar ao simulador o momento de detectar ou disparar eventos, tais como ondas recebidas através do rádio.

3.3. Ferramenta de desenvolvimento *MAKE*

Make é uma ferramenta que controla a geração de executáveis e outros arquivos não fonte de um programa a partir de arquivos fonte do programa (*GNU Fundation*), ou seja, caso ocorra alguma alteração em um código que seja dependente direto ou indireto de outro de prioridade maior, o *Make* compila somente os arquivos necessários, evitando a recompilação desnecessária de todo o programa.

Só que para isso, o programador deve criar um arquivo *Make* (chamado *Makefile*) e através de diretivas ou regras, escrever as dependências do arquivo. Através dessas diretivas, a ferramenta *Make* busca o arquivo alvo e executa as regras inseridas no *Makefile* para obter uma consistência no sistema resultante. Neste arquivo, o alvo sempre se localiza na coluna esquerda antes do caractere ‘:’, na coluna da direita estão os arquivos cujo o alvo depende, essa linha onde o alvo se encontra, é chamada de linha de dependência.

Logo abaixo a essa linha, deve-se inserir os comandos que serão as regras que devem ser obedecidas quando a ferramenta executar ou quando existir alguma alteração ou atualização de seus dependentes.

No caso desse trabalho, a ferramenta é utilizada para auxiliar na geração e gerência dos códigos de cada protocolo simulado.

3.4. *Tmote Sky*

As redes de sensores sem fio requerem dispositivos de baixo consumo de energia, comunicação de rádio, processamento e memória suficientes para o funcionamento estável dos nós. O *Tmote Sky* é um módulo sem fio de baixíssimo consumo de energia para uso em redes de sensores sem fio, monitoramento de aplicações e protótipos de aplicações rápidas. Este utiliza os padrões da indústria, como o USB e IEEE 802.15.4 para interagir com outros elementos da rede (TMOTE SKY, 2006). Para coletar dados, o *Tmote Sky* possui sensores de temperatura, umidade e luz integrados.

A plataforma descrita pode ser observada na figura 3, esta é alimentada por duas pilhas AA quando não estão conectados em uma porta USB, quando conectados, são alimentados através da porta USB. Por ser necessário economizar energia, esta plataforma desliga o rádio quando o mesmo não está sendo utilizado, já que este é o elemento de maior consumo energético em toda plataforma.

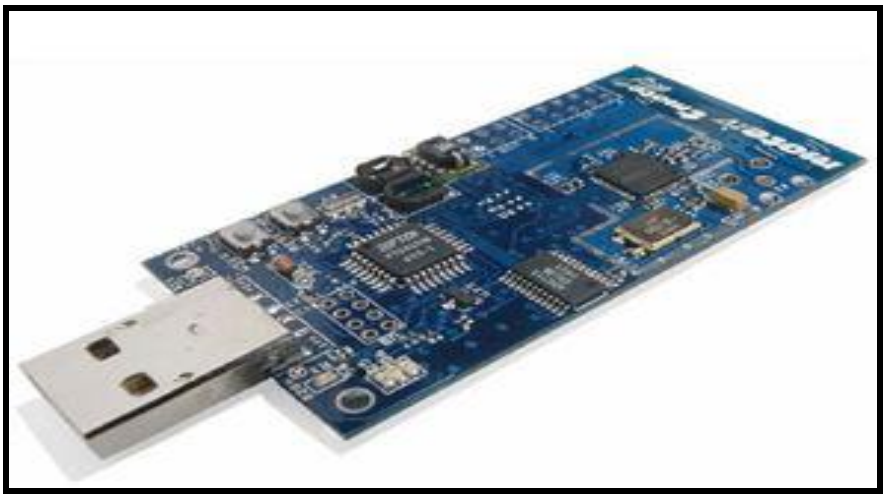

Figura 3: ***Tmote Sky***

Os principais recursos desta plataforma são:

- Transceptor wireless chipcon IEEE 802.15.4, 2.4Ghz, 250 kbps;
- Interoperabilidade com outros dispositivos IEEE 802.15.4;
- Microcontrolador de 8Mhz, MSP430 da *Texas Instruments* (10k RAM, 48k Flash);
- Conversores ADC e DCA integrados, supervisor de fonte de tensão e controlador DMA;
- Antena *on-board* integrada com 50 metros de alcance em ambientes fechados e 125 metros de alcance ao ar livre;
- Sensores *on-board* integrados de umidade, temperatura e luz;
- Criptografia e autenticação de hardware na camada de enlace;
- Programação e coleta de dados via USB;
- Suporte de expansão de 16 pinos e conector para antena SMA opcional;
- Suporte *TinyOS*: rede de malha e implementação de comunicação.

Com esta gama de recursos é possível uma ampla faixa de aplicações com o *Tmote Sky*, uma delas foi o uso na simulação da pilha de protocolos *uIP* e IPV6. O COOJA oferece um simulador desta plataforma, assim, por não dispor do dispositivo físico, foi utilizado o dispositivo virtual oferecido pelo COOJA nas simulações deste trabalho.

4 TRABALHOS RELACIONADOS

Neste capítulo são expostos trabalhos relacionados à simulação da Internet das Coisas, priorizando a exposição de trabalhos que possuem como foco principal simular de qualquer modo a *IoT*.

Um trabalho relevante neste âmbito é o trabalho de Michael Kirsche (KIRSCHE, 2013), neste trabalho o autor propõe um ambiente de simulação híbrido que tem como objetivo realizar estudos de simulação da Internet das Coisas. E ele tem como principal interesse, simular corretamente o nó em nível de sistema e em nível de rede ao mesmo tempo, fornecendo uma simulação que permite testar seus protocolos de aplicação durante a fase de pré-implantação. Visto que os simuladores COOJA (ÖSTERLIND et al, 2006) e OMNeT++ (VARGA et al, 2008) possuem pontos fortes em suas respectivas áreas, porém não cobrem o nó a nível de sistema e nível de rede ao mesmo tempo, o autor propôs este ambiente para reduzir a lacuna entre a pesquisa e o desenvolvimento prático.

De outra maneira, o trabalho de ÖSTERLIND et al (2006) também está relacionado com o tema central deste trabalho, pois ele apresenta o simulador COOJA para simular uma rede de sensores sem fio. Este foi o simulador escolhido para este trabalho, principalmente devido a sua simulação de forma cruzada, permitindo a simulação simultânea em vários níveis do sistema.

Sabendo que RPL (*IPv6 Routing Protocol for Low Power and Lossy Networks*) é o padrão candidato IETF para roteamento IPv6 em redes de sensores sem fio de baixa potência, os primeiros resultados que foram obtidos com a implementação do ContikiRPL são apresentados em TSIFTES et al (2010). O ContikiRPL é uma implementação do protocolo de roteamento RPL para baixo consumo de energia e as redes com perdas. Contudo, segundo os autores, os estudos sobre RPL comprovaram que experiência prática de implementações em sistemas com recursos limitados, semelhantes aos objetos da *IoT*, tem faltado. Por isso, os autores desenvolveram e implementaram o RPL dentro da pilha *µIP* com IPv6, realizando experimentos tanto em uma rede sem fio de baixa potência quanto na simulação.

Como pode ser visto, não foi encontrado nenhum trabalho que simulasse a pilha *µIP* por completo com o protocolo IPv6, levando assim, a necessidade da sua simulação e da publicação dos resultados na academia, para servir de base didática para o estudo e para o desenvolvimento de novas aplicações.

5 SIMULAÇÃO EM *IoT*

Simulação é, em geral, entendida como a “imitação” de uma operação ou de um processo do mundo real. A simulação envolve a geração de uma “história artificial” de um sistema para análise de suas características operacionais (MIYAGI P. E., 2006).

A simulação pode ser usada de várias formas de acordo com as necessidades, em alguns casos, é usada para prevê o comportamento de um sistema que será utilizado, em outros é usada para avaliar e possivelmente validar o desempenho de um sistema já existente.

Existem outros meios de prever o comportamento de sistemas como, por exemplo, os modelos matemáticos desenvolvidos através de equações, porém em muitos casos em que o sistema é muito complexo, modelos matemáticos se tornam extremamente difíceis de serem desenvolvidos, fortalecendo assim a ideia da utilização de simulação. Devido o avanço da computação, novas ferramentas de simulação poderosas foram desenvolvidas, fazendo da simulação uma técnica muito usada no desenvolvimento ou avaliação de sistemas.

Os modelos de simulação dinâmicos podem ser discretos ou contínuos, considera-se uma simulação discreta quando o tempo entre os eventos ocorridos não é importante para o resultado final da simulação, o qual também é chamado de simulação de eventos discretos. Já na simulação contínua, o tempo é de fundamental importância no resultado.

Nas simulações da pilha *uIP* e da pilha Rime com IPV6, o tempo não altera o resultado da simulação, portanto foram feitas simulações discretas através do COOJA, o qual oferece uma caixa de saída chamada *outline*, que mostra os eventos que ocorreram durante a simulação, tais eventos são descritos discretamente em linhas designadas para cada nó na rede.

5.1 Planejamento da Simulação

A quantidade de simulações realizadas depende da natureza do protocolo utilizado, a priori foi decidido padronizar, porém no decorrer do desenvolvimento percebeu-se a necessidade de adaptar a quantidade de simulações às métricas fornecidas por cada sistema. Devido a isso, a quantidade de simulações realizadas foi:

- *Hello world* - Uma simulação;
- *Simple* UDP - Quatro simulações;
- UDP - Quatro simulações;

- TCP - dezessete simulações;
- REST - Quatro simulações;
- Rime - Duas simulações.

- **Métricas**

Para avaliar o desempenho das simulações foram observadas as seguintes métricas:

- Pacotes enviados (IP, TCP, UDP);
- Pacotes Recebidos (IP, TCP, UDP);
- Pacotes Perdidos (IP, TCP, UDP);
- Pacotes Retransmitidos (TCP);
- RSSI (*Received Signal Strenght Indication*);
- Estatística de energia (Rime);
- Estatística oferecida pela estrutura rimestats (Rime).

RSSI é um valor que indica a intensidade do sinal de rádio recebido, ou seja, é uma indicação relativa sobre a qualidade da conexão entre o receptor e o emissor. Esse valor pode variar de zero a duzentos e cinquenta e cinco, mas essa faixa não é padronizada, cada empresa fornece uma escala individual. No caso da *Texas Instruments*, o transceptor CC2420 (*ZIGBEE-READY RF TRANSCEIVER*) varia de zero a menos cem (0 a -100) dBM (decibel *miliwatt*) e possui um valor de *offset* de menos quarenta e cinco em relação ao valor obtido no registrador.

Porém, por se tratar da potência do sinal que é recebido, quanto maior o valor, melhor o sinal. O qual pode ser confirmado pela estrutura do RSSI, pois ele é o resultado da qualidade da antena do receptor (potência fornecida) e do emissor (sensibilidade ao sinal), e da perda e atenuação do sinal pelo caminho transitado.

De certa forma a distância entre o emissor e o receptor interfere no valor de RSSI, mas segundo PARAMESWARAN et al (2009) o RSSI não pode ser utilizado como uma métrica confiável para medir distâncias, por não ter um comportamento consistente devido à interferência de outros objetos e atenuação do sinal.

Além das métricas citadas, também foram analisados parâmetros que influenciam o desempenho do sistema. Os parâmetros que foram variados durante a simulação (fatores) são:

- Disposição dos nós (fixos ou aleatórios);
- Taxa de sucesso de transmissão (TX);
- Taxa de sucesso de recepção (RX).

E os seus possíveis valores (níveis):

- Níveis de TX e RX: 100% e 50% para todas as simulações exceto TCP;
- Níveis de TX para TCP: 100%, 80%, 70%, 60%, 50%;
- Níveis de RX para TCP: 100%, 90%, 80%, 70%, 60%, 50%, 40%, 30%, 20%, 10%.

Em resumo, com o objetivo de analisar a relação entre o número de nós e a taxa de sucesso de TX/RX com a contagem de pacotes, alterou-se o número de nós e a porcentagem da eficiência do sinal em cada simulação. Também definiu como padrão o valor do alcance de transmissão de cada nó em 50 metros, o alcance de interferência em 100 metros e o tempo de simulação em uma hora (tempo simulado).

Para alterar a taxa de sucesso clica-se com o botão direito em qualquer espaço vazio na janela *network* e efetua a mudança do valor de TX e RX para cinquenta por cento. Além de obter todas as métricas em um mesmo nó, também se utilizou o método de aleatoriedade padrão do simulador COOJA para a disposição dos nós.

5.2 Montagem do Ambiente

As simulações foram executadas no sistema operacional Contiki através do *InstantContiki*, um ambiente de desenvolvimento completo com todas as ferramentas de desenvolvimento, compilação e simulação Contiki necessárias. Neste trabalho foi utilizado o simulador COOJA, presente naquele, com a plataforma virtual *Tmote Sky*.

1. Para instalar o *software* faça o *download* do *InstantContiki* 2.6.1 no endereço: http://sourceforge.net/projects/contiki/files/Instant%20Contiki/Instant%20Contiki%202.6.1/InstantContiki2.6.1.zip/download.
2. Faça o download de uma máquina virtual, a utilizada neste trabalho está disponível no endereço: https://my.vmware.com/web/vmware/free#desktop_end_user_computing/vmware_player/5_0.

3. Após fazer ambos os *downloads*, descompacte o *InstantContiki* colocando-o na área de trabalho do usuário que for utilizar o mesmo, pode-se escolher um outro lugar, porém neste caso deve-se mostrar o caminho até o local onde a imagem foi descompactada.
4. Abra a máquina virtual e execute a imagem do *InstantContiki* 2.6.1 como mostrado na figura 4.

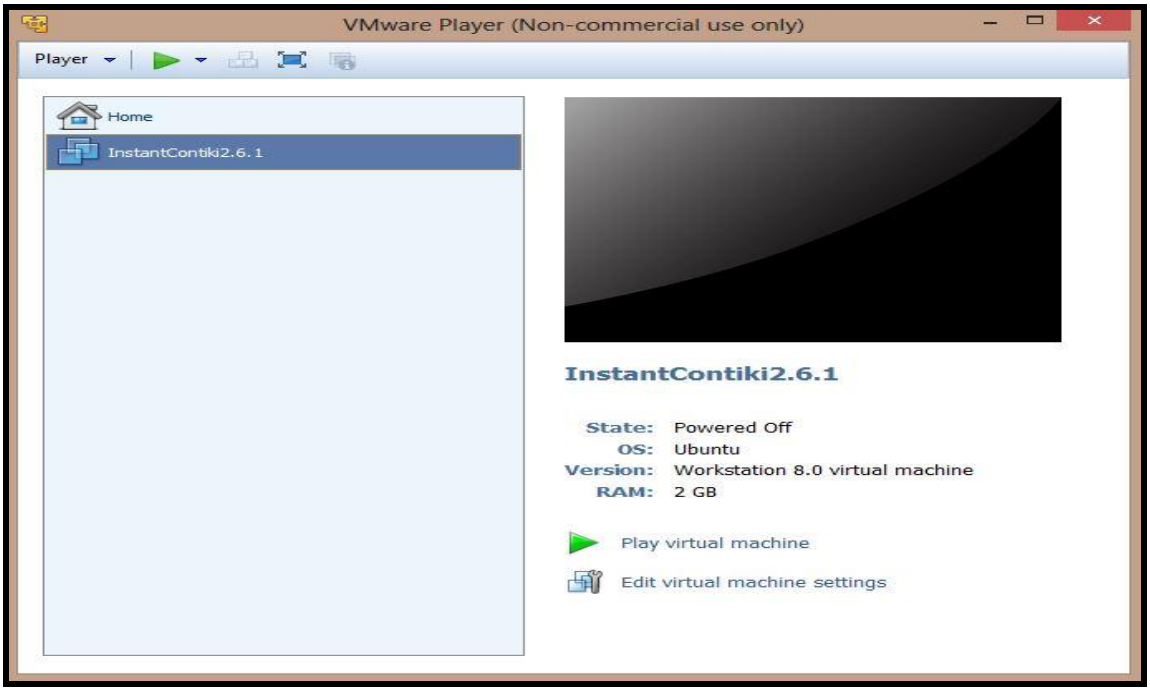

Figura 4: Executando VMware.

1. Faça o login utilizando a senha *user* na tela de inicio da máquina virtual como mostra a figura 5.

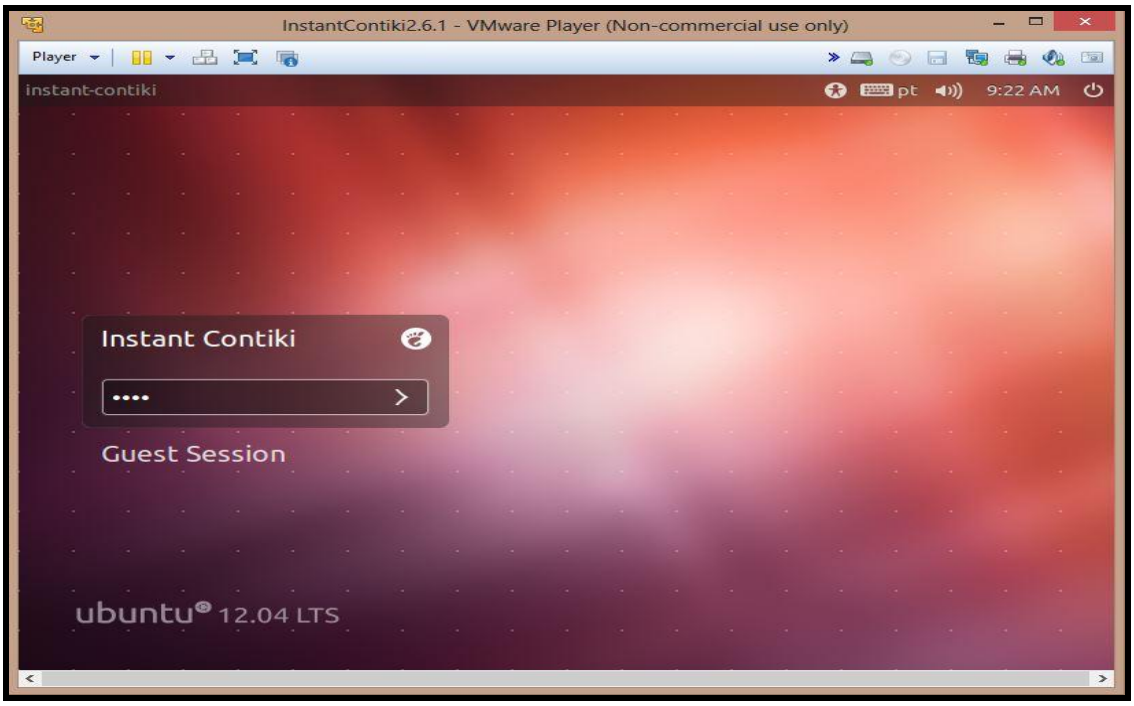

Figura 5: Tela de Início

2. Para criar o primeiro projeto Contiki é necessário um arquivo fonte em linguagem de programação C e um *Makefile*, o qual contém as regras para compilar o código do programa executável Contiki. Uma simples simulação será mostrada para ilustrar o ambiente.

- Crie dois diretórios, um para projetos e outro para o projeto com título de *hello world*. Os diretórios podem ser criados normalmente com o botão direito do mouse e clicando em nova pasta ou através de linhas de comando. A figura 6 mostra tais comandos de criação.

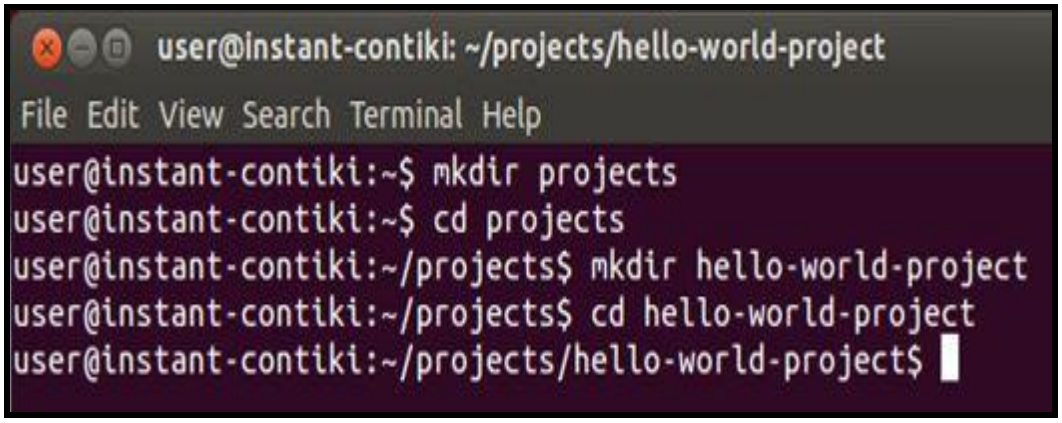

Figura 6: Criando diretórios.

- Crie um arquivo *Makefile* em um editor de texto, pode-se usar o *gedit*, e adicione as linhas:

- CONTIKI=/home/user/contiki
- Include $(CONTIKI)/Makefile.include

A figura 7 apresenta o arquivo *Makefile* criado.

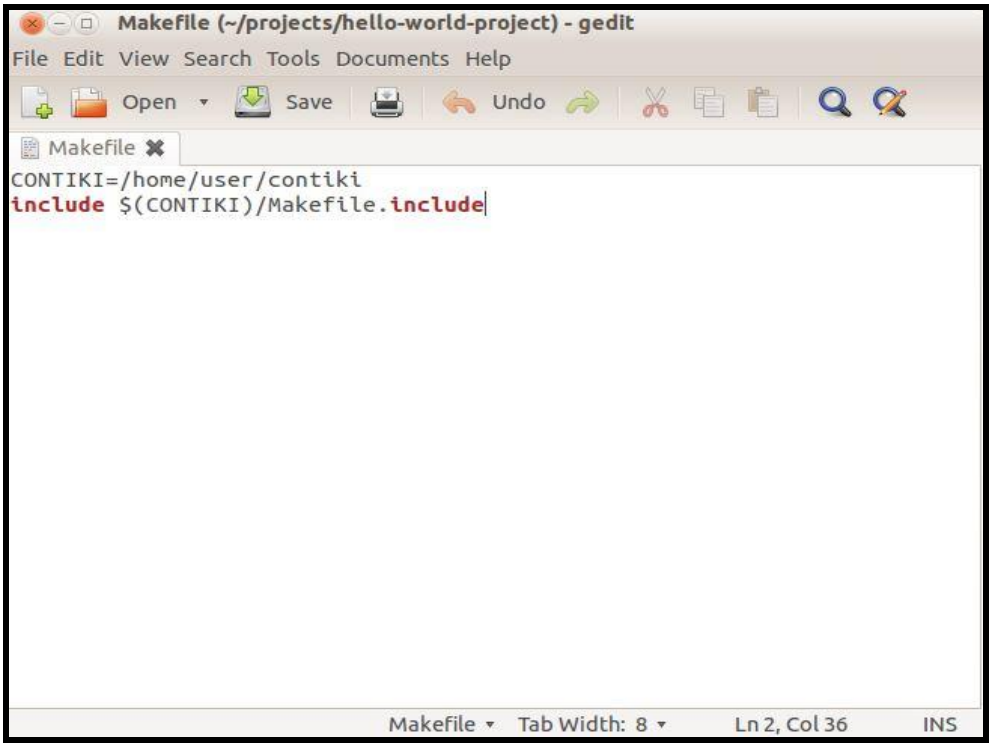

Figura 7: Arquivo *Makefile*.

Agora é preciso um arquivo fonte em linguagem de programação C com o código para exibição do *Hello World* ilustrado na figura 8, é necessário adicionar a biblioteca de entrada e saída do C para o uso da função *printf* e a biblioteca do Contiki para que possam ser chamados as funções e macros pertencentes ao mesmo.

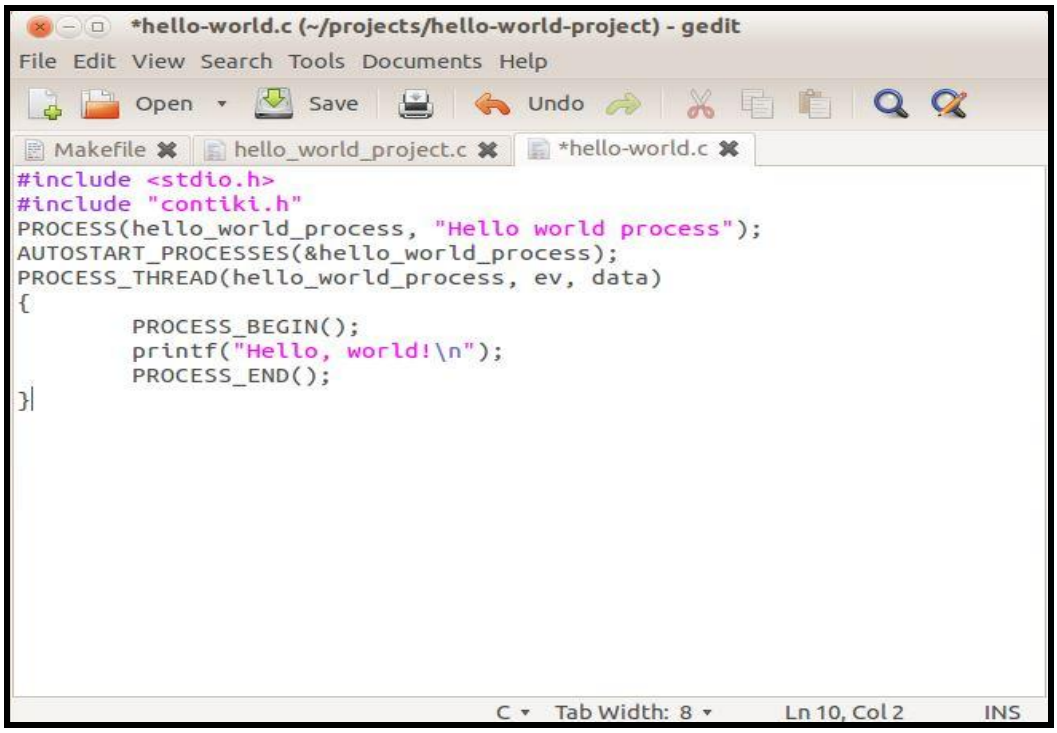

Figura 8: Arquivo fonte *Hello World*.

3. Com o primeiro projeto pronto, já é possível simular nós exibindo a mensagem *Hello World*, para isso clique no ícone do COOJA na área de trabalho ou abra através do terminal, como mostra a figura 9, com o comando:

- cd contiki-2.6/tools/cooja/
- ant run

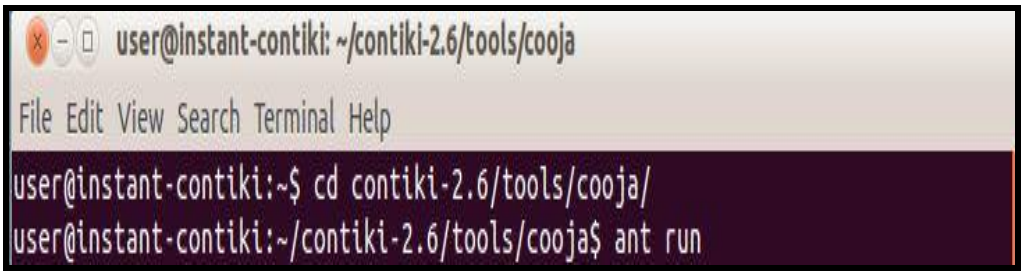

Figura 9: Executando COOJA.

4. Na tela observada na figura 10, selecione *File -> New Simulation*, digite o nome da simulação e clique em *Create.*

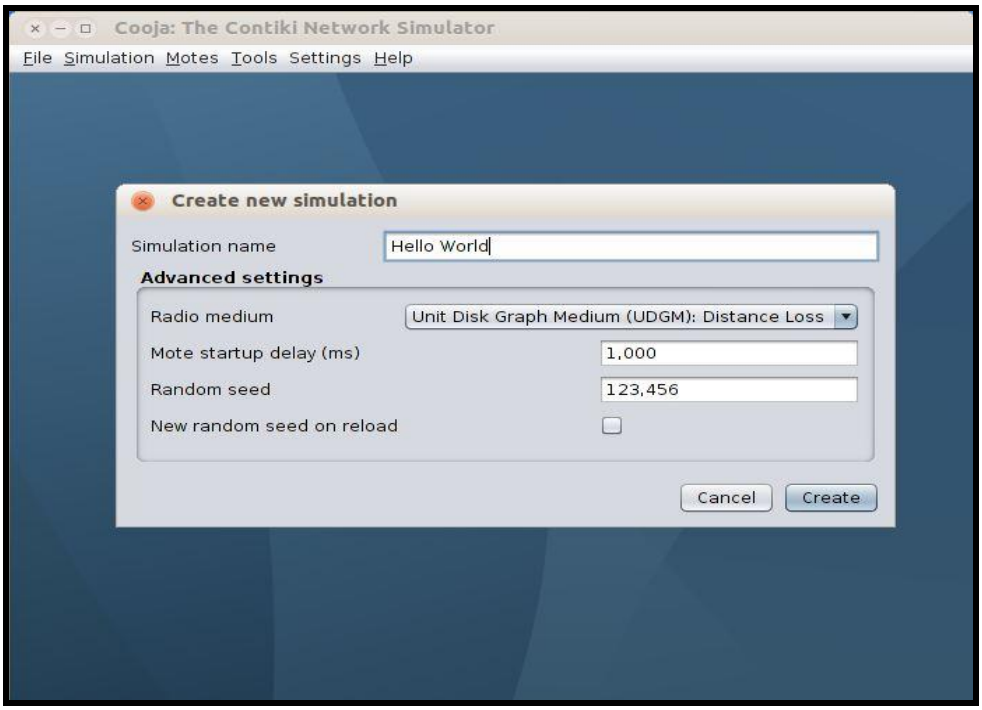

Figura 10: Criando nova simulação.

5. Na nova área que irá aparecer, como mostra a figura 11, selecione *Motes -> Add motes -> Create new mote type -> Sky mote*.

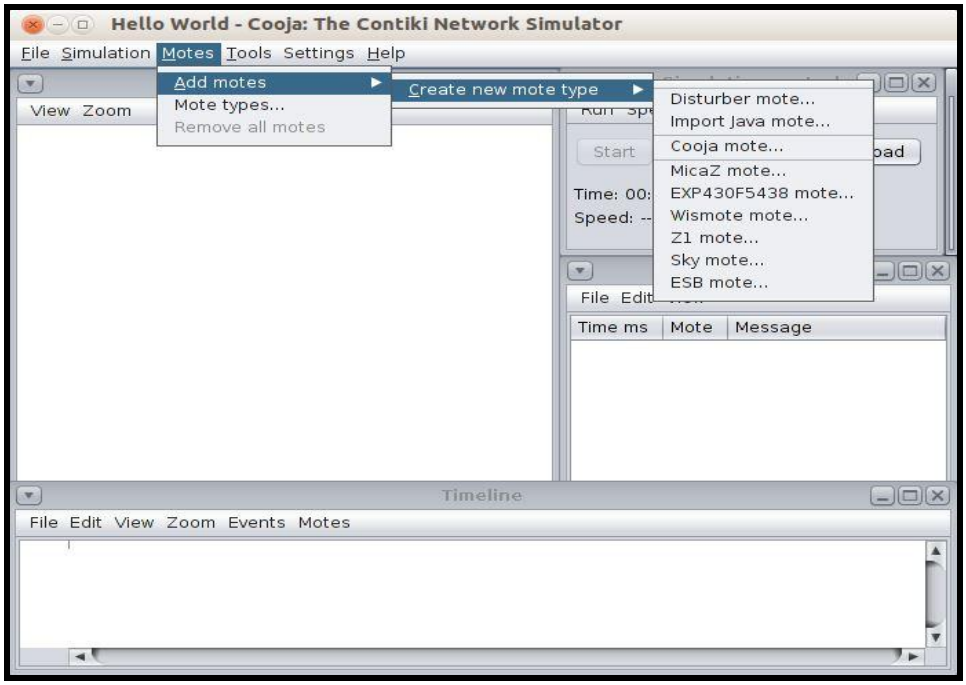

Figura 11: Adicionando *Sky mote*.

6. Abrirá uma janela semelhante à figura 12, clique em *Browse* e adicione o seu arquivo .c para compilação.

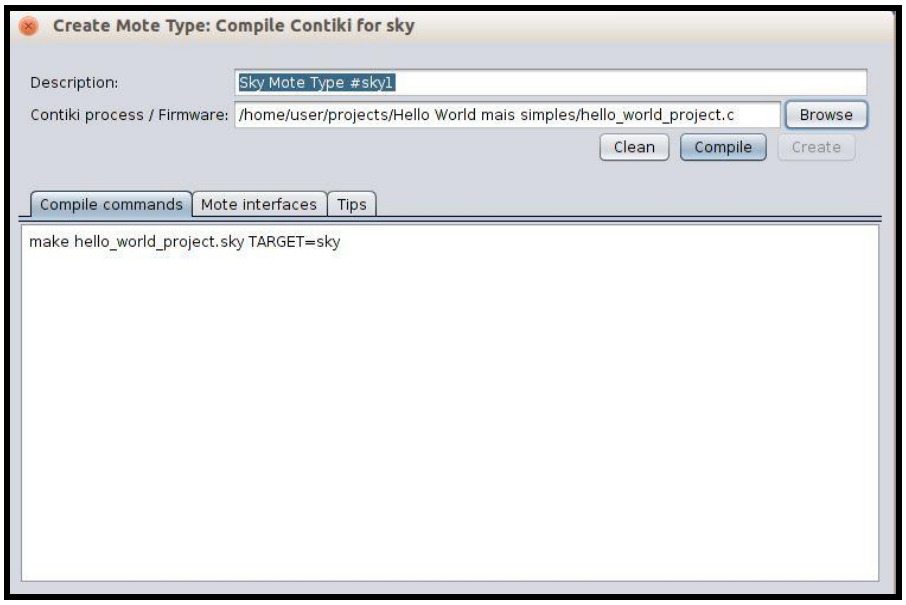

Figura 12: Adicionando arquivo c para compilação.

7. Clique em *Compile* e espere a compilação terminar, se houver erros serão descritos na aba *compilation output* que aparecerá assim que iniciar a compilação.

8. Clique em *Create* e escolha o número de nós, suas posições e clique em *Add motes*. Neste caso deixaremos as posições aleatórias e criaremos apenas um nó, como exibido na figura 13.

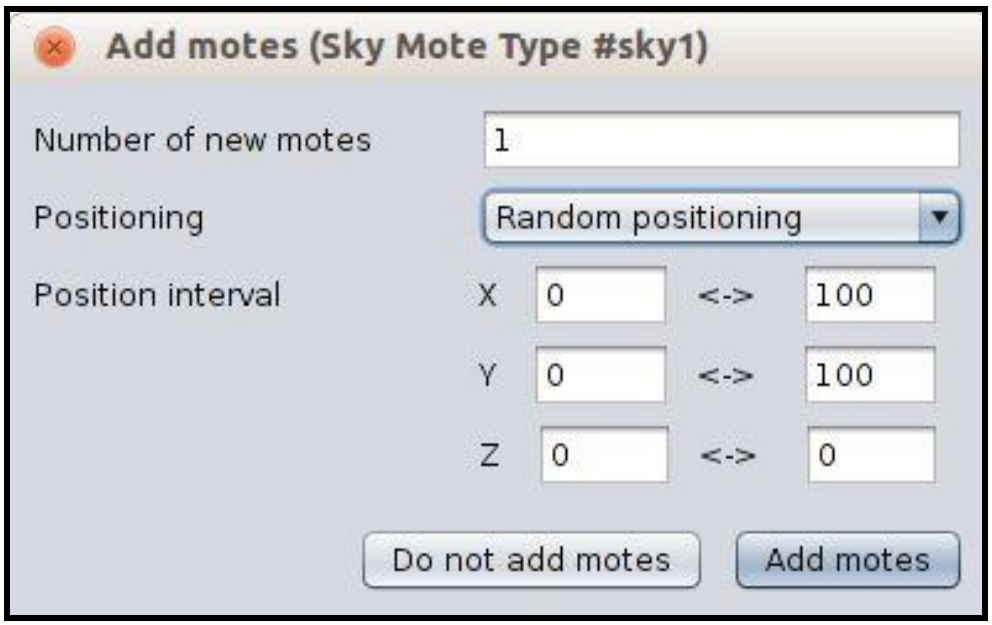

Figura 13: Adicionando nós.

9. Em *simulation control* selecione start e observe a janela *mote output*, esta mostrará os registros de saídas de cada nó simulado.

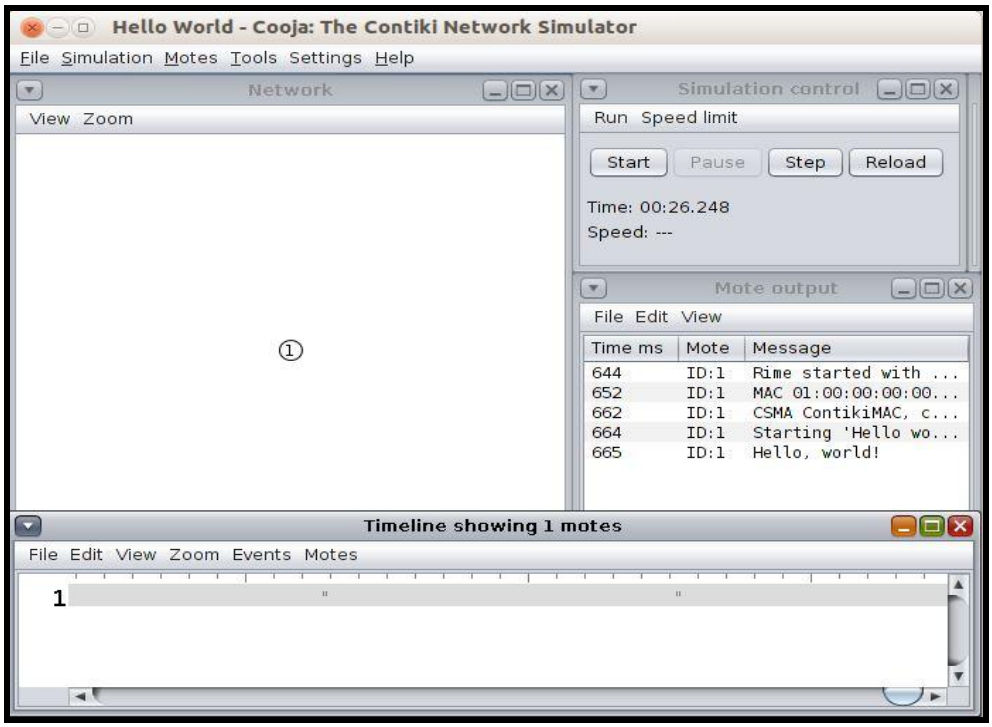

Figura 14: Tela do COOJA executando a simulação *Hello World*.

10. A figura 14 mostra a área de trabalho do ambiente do simulador COOJA, nele é possível observar:

- *Network*: Local onde os nós são distribuídos representando a rede criada. Em *View* é possível selecionar diferentes modos de visualização dos nós, o que visa facilitar a observação dos eventos durante a simulação.
- *Simulation Control*: Controla a velocidade, início, fim e pausas na simulação. Caso a simulação use eventos de tempo é aconselhável que mantenha a velocidade de simulação em 100%.
- *Mote Output*: É o registro de todos os eventos que ocorreram na simulação, nesta janela é possível ver todas as saídas colocadas no código em linguagem de programação C. No menu desta janela pode-se selecionar filtros de acordo com o desejado, exportar o registro para um arquivo de texto, colorir as mensagens por nó facilitando a visualização, limpar as mensagens, entre outros.
- *TimeLine*: A linha do tempo mostra os eventos de simulação ao longo do tempo. Esta pode ser usada para inspecionar as atividades dos nós individuais, bem como as interações entre eles.

- Além da área de trabalho básica descrita na figura 14, clicando em *Tools* percebe-se que existem várias outras janelas que podem ser observadas de acordo com a necessidade do projeto.

5.3 Simulações

Neste subtópico são apresentadas todas as simulações realizadas neste trabalho e os resultados obtidos. A apresentação é dada na seguinte ordem: *Hello World*, *Simple* UDP, API UDP, Rime, TCP e REST.

5.3.1 Hello World:

Nesta simulação é apresentado um código mais robusto do *Hello World*, o qual tem como objetivo utilizar alguns dos periféricos disponíveis no *Tmote Sky* como, por exemplo, *leds* e botões além de estruturas de controle baseadas em tempo (chamadas de eventos de *timer*).

Para utilização dos periféricos é necessário a inclusão da biblioteca que implementa os mesmos. A biblioteca “button-sensor.h” é responsável pelas funções e macros relacionadas ao botão do dispositivo virtual, que pode ser acionado clicando com botão direito do mouse em cima do nó e selecionando *Click button on sky x* (x é o numero do nó em questão). As funções e macros que controlam os *leds* são introduzidas pela biblioteca “leds.h”.

Com o intuito de demonstrar os periféricos citados, assim como o controle de tempo, esta simulação primeiramente exibe uma mensagem de *Hello World* e fica esperando um clique no botão, assim que pressionado é disparado um *timer* de 5 segundos e os *leds* verde e vermelho são alternados como se observa na figura 15.

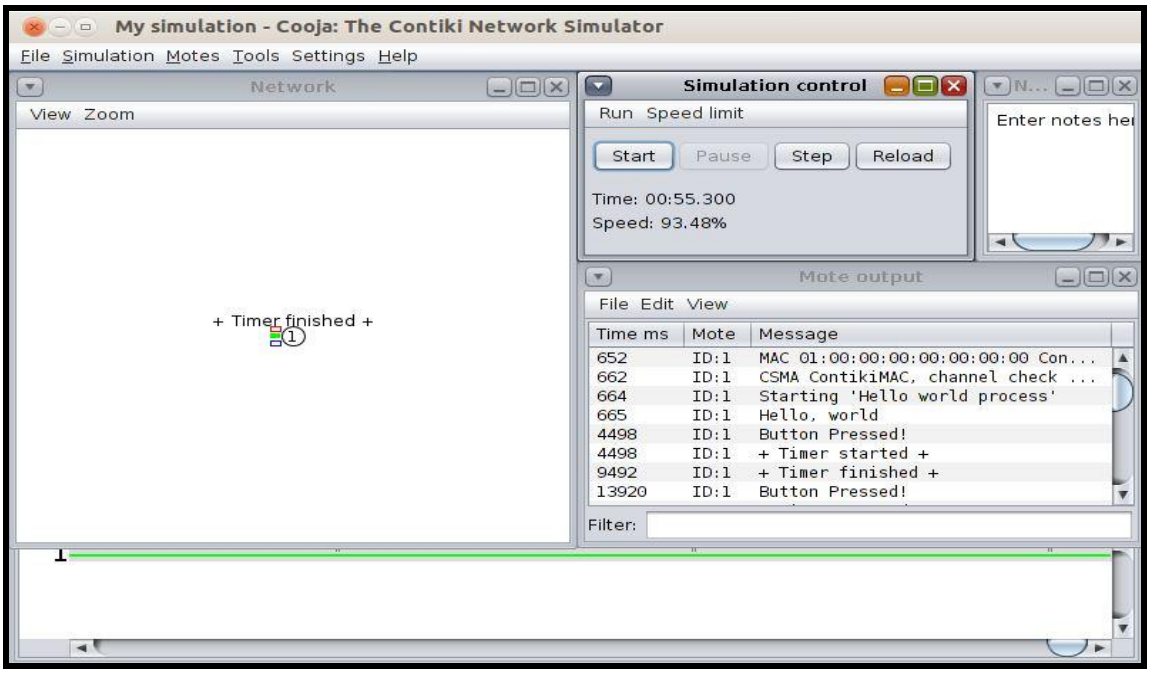

Figura 15: Simulação *Hello World* com utilização de dispositivos.

- Resultados obtidos:

O objetivo da simulação *Hello World* foi apenas mostrar o uso de periféricos oferecidos no *Tmote sky* como *leds* e botões, além da utilização de eventos baseados em tempo.

Assim, não houve comunicação entre dois ou mais nós via rádio, o que impossibilitou a coleta de estatísticas nesta simulação.

5.3.2 *Simple* UDP:

Devido à dificuldade de usar a API UDP comum do Contiki, o mesmo oferece uma API mais simples chamada de *Simple* UDP, a qual disponibiliza três funções básicas:

- Simple_udp_register: registra uma conexão UDP e anexa uma função de *call-back* que é chamada sempre que chegar pacotes.
- Simple_udp_send: envia um pacote UDP para conexão que foi registrada.
- Simple_udp_sendto: envia um pacote UDP para um IP específico.

Com o objetivo de simular a pilha de protocolos *uIP* com comunicação IPv6, foi escrito um código em linguagem de programação C que permite um nó mandar pacotes em *broadcast* para os demais nós na rede, esta simulação é totalmente escalável quanto ao número de nós.

Para isso foi utilizado a função de registro com intuito de registrar uma conexão UDP, e em seguida uma função chamada uip_create_linklocal_allnodes_mcast(&addr) que é implementada no arquivo uip.c do Contiki. Esta função permite que cada nó colocado na simulação procure os demais nós e obtenha seus respectivos endereços automaticamente. Com os nós devidamente endereçados, foi utilizada a função simple_udp_sendto para que cada nó envie um pacote UDP para os demais na rede formando um *broadcast*.

Para coletar as estatísticas dos pacotes enviados, recebidos, perdidos e o valor do RSSI, utilizou-se a estrutura implementada pelo uip.c chamada uip_stats, esta conta automaticamente os pacotes recebidos, enviados e os perdidos, podendo estes terem sido perdidos por diversas formas, sendo contados tanto os pacotes IP como os pacotes UDP.

Para que a estrutura uip_stats funcione, é necessário que seja modificado o código fonte do uip.h e o uip.c. Esta alteração deve ser feita na seguinte linha:

- Na linha onde se encontra #if UIP_STATISTICS == 1 trocar por #if UIP_STATISTICS. A alteração deve ser feita em ambos os arquivos uip.c e uip.h para que a coleta de estatísticas funcione.

Outro procedimento de fundamental importância é a alteração no *Makefile*, este deve conter as informações de configuração para uso da pilha *uIP*, para ativar o IPv6, como para o uso da coleta de estatísticas. O *Makefile* resultante está ilustrado na figura 16.

Makefile

```
CONTIKI=/home/user/contiki/
WITH_UIP=1
UIP_CONF_IPV6=1
UIP_CONF_STATISTICS=1
include $(CONTIKI)/Makefile.include
```

Figura 16: Makefile Simple UDP.

Por fim foi adicionado um *timer* de 3 segundos com objetivo de obter uma melhor visualização dos eventos ocorridos durante a simulação, esta é mostrada na figura 17.

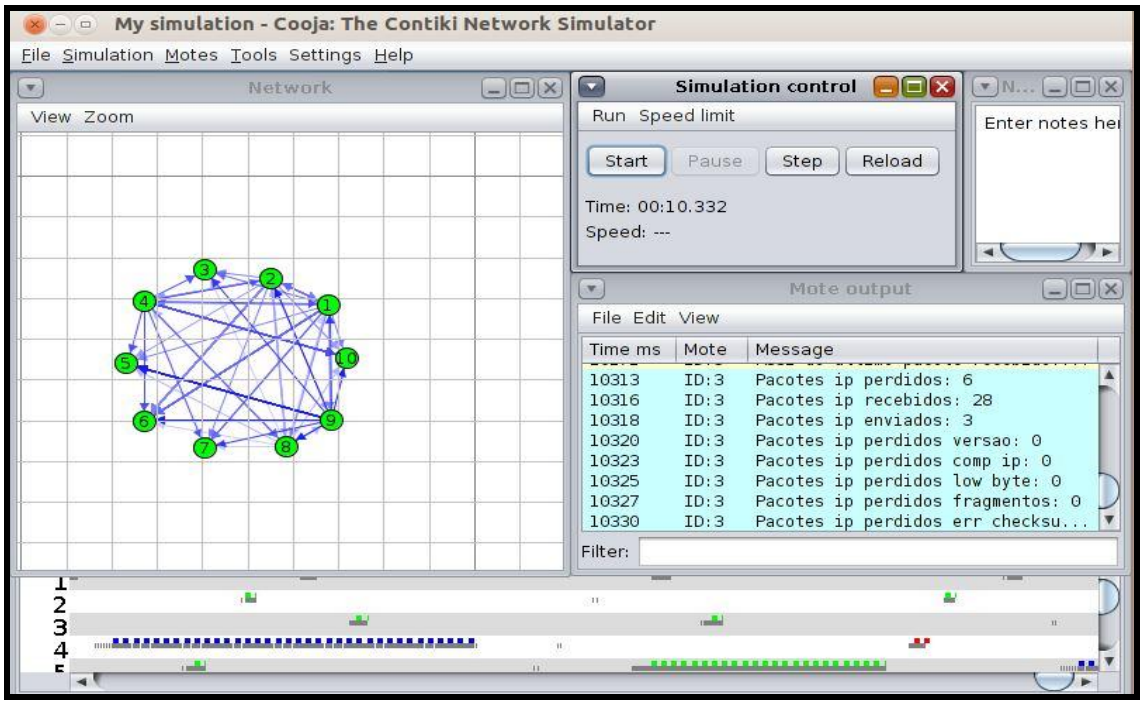

Figura 17: Simulação *Simple* UDP.

- Resultados obtidos:

O nó número um foi selecionado para análise das métricas.

- Relação nós – Contagem de pacotes: Realizou-se uma simulação com dois nós e logo em seguida com dez nós, respeitando uma hora (tempo simulado) de cada simulação e o método de aleatoriedade padrão do COOJA para disposição dos nós. Os valores obtidos estão apresentados nas tabelas 1, 2 e 3.

Tabela 1 - Relação de nós com pacotes IP.

Nº nós	Perdidos	Recebidos	Enviados	Perdidos por erro de cabeçalho	Perdidos por erro no comprimento: *High byte*	Perdidos por erro no comprimento: Low byte	Perdidos devido a Fragmentação IP.	Perdidos Erro de checksum	Perdidos por não serem TCP, UDP ou ICMP
02	1	19	19	0	0	0	0	0	0
10	4	66	19	0	0	0	0	0	0

Tabela 2 - Relação nós com datagramas UDP.

Nº nós	Perdidos	Recebidos	Enviados	Perdidos Erro de checksum
02	0	18	19	0
10	0	62	19	0

Tabela 3 - Relação nós com o RSSI.

Nº nós	Base RSSI (dBm)	RSSI do último pacote (dBm)
02	-89	-89
10	-100	-89

Nos valores obtidos na tabela 1, a qual mostra a relação dos nós na camada IP, é possível perceber que os pacotes recebidos e enviados tiveram o mesmo número quando existiam apenas dois nós, com dez nós em simulação foram obtidos o mesmo número de pacotes enviados, já que ambas as simulações tiveram o mesmo tempo, porém poucos nós recebidos, se comparado com a situação de dois nós. Isso se dá devido ao fato dos nós estarem posicionados aleatoriamente na área de trabalho com a distância máxima da antena de 50 metros, assim o nó em análise (nó um) não é alcançado por todos os demais nós, recebendo o pacotes apenas dos nós que o alcançam.

Ainda nesta tabela, é possível notar que a quantidade de pacotes perdidos na camada de rede (IP) teve um leve aumento, isso era esperado já que aumentou o tráfego na rede. A função *uip_stats* descrita nas simulações fornece alguns possíveis erros para perda de pacotes IP, porém em todas as simulações feitas os pacotes perdidos por estes erros foram sempre zero.

A tabela 2 mostra a relação dos datagramas na camada de transporte utilizando protocolo UDP, nesta nota-se a ausência de datagramas perdidos e que a relação de datagramas enviados e recebidos teve um comportamento similar ao observado na camada de rede.

Em relação ao RSSI a tabela 3 mostrou que em ambos os casos o comportamento foi similar, com uma pequena queda na base quando ocorreu o aumento de nós, o que já era esperado devido o aumento do tráfego na rede.

Para analisar os datagramas UDP enviados e recebidos em todos os nós, foi executada uma simulação com dez nós variando a posição dos nós e modificando a taxa de sucesso na transmissão e recepção no rádio do nó.

Dispondo os nós aleatoriamente no COOJA como mostra a figura 18, obteve-se picos de pacotes recebidos, fato este que ocorre nos nós 2 e 9, estes picos podem ser vistos no gráfico da figura 19. Isso ocorre devido o posicionamento desses nós serem estratégicos em comparação aos demais, como eles se encontram em locais que a maioria dos outros nós conseguem alcançá-los, eles tendem a receber mais pacotes. Como já citado acima, cada nó somente envia pacote até um raio máximo de 50 metros, sendo assim os pacotes recebidos dependem dos demais nós alcançarem ou não o nó em questão. Isso pode ser confirmado através do cenário montado da figura 20, este foi montado com os nós próximos um dos outros de tal forma que todos os nós consigam enviar para os demais.

Em tal cenário cada nó enviou 19 pacotes, cada nó recebe 1 pacote dos outros 9 em cada instante (determinado por um *timer* de 3 segundos), então cada nó poderia receber no máximo 171 pacotes UDP caso estivesse em um ambiente que não fosse possível perda de pacotes. Observa-se no gráfico da figura 21, que com os nós próximos a ponto de todos conseguirem se comunicar entre si os mesmos conseguiram chegar próximo da totalidade (171 pacotes). O fato de não receberem a totalidade dos pacotes mostra que houve algumas perdas de pacotes, a simulação exibiu que alguns desses pacotes foram perdidos na camada de rede.

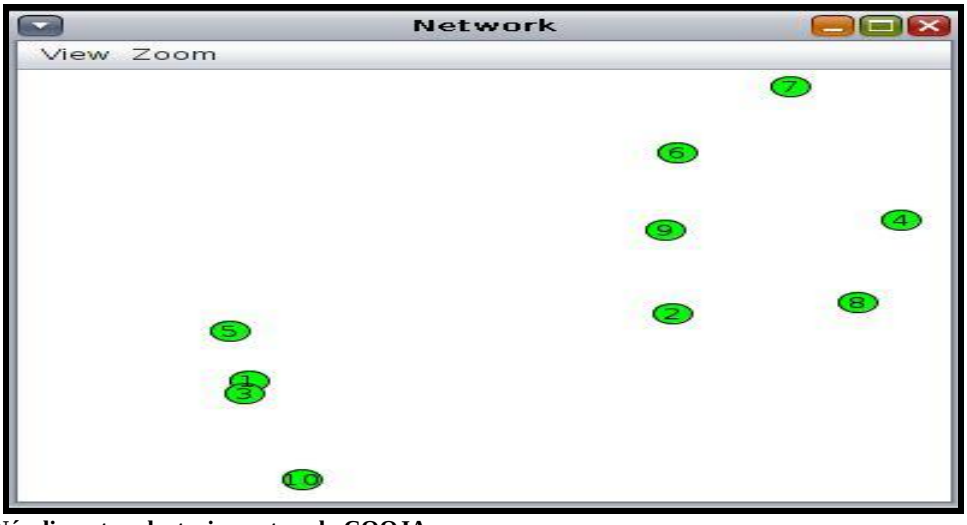

Figura 18: Nós dispostos aleatoriamente pelo COOJA.

Figura 19: Histograma dos pacotes enviados e recebidos com os nós espalhados aleatoriamente pelo COOJA.

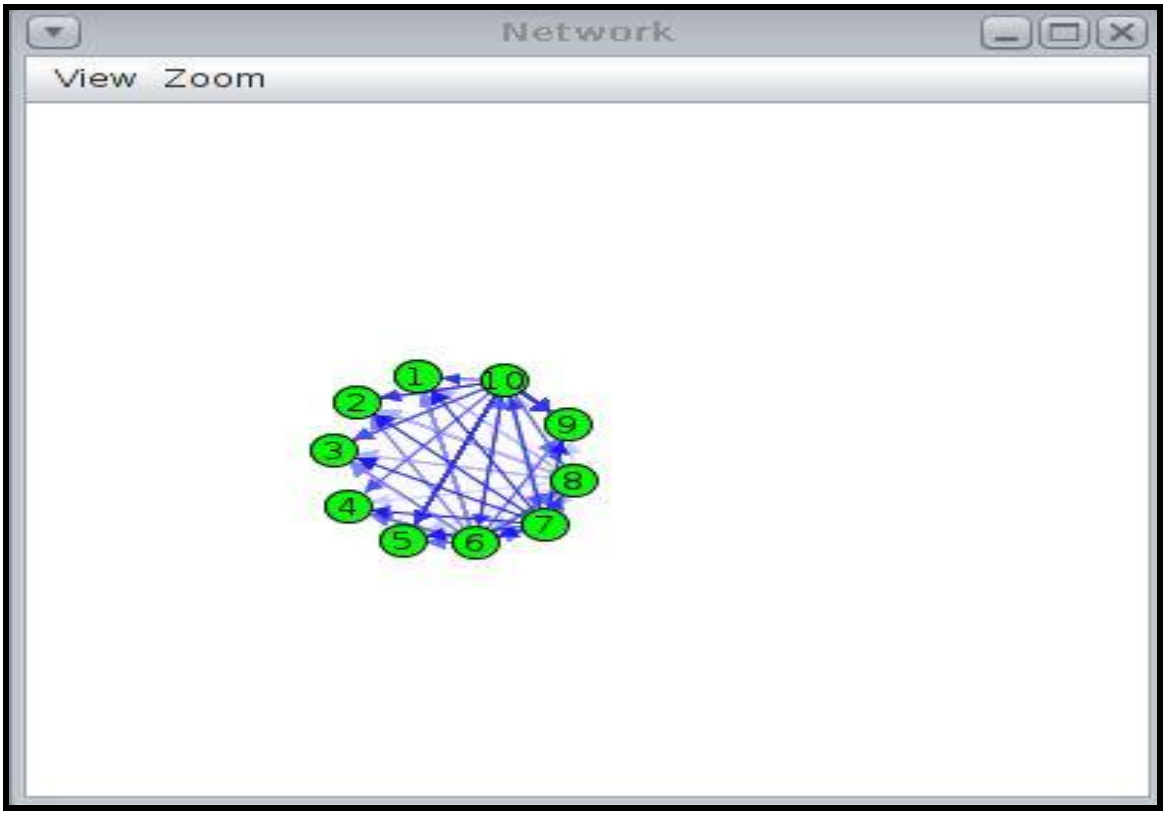

Figura 20: Nós dispostos em pontos fixos.

Figura 21: Histograma dos pacotes enviados e recebidos com os nós em pontos fixos.

- Relação nós – TX (Taxa de sucesso de transmissão) e RX (Taxa de sucesso de recepção): Nesta parte é alterada a taxa de sucesso de transmissão e recepção e testado nos cenários com nós dispostos aleatoriamente e nos cenário com nós dispostos em pontos fixos.

A simulação foi executada com 50% do TX e 50% de RX, os resultados obtidos para os cenários com nós aleatórios e em pontos fixos podem ser analisados através dos gráficos das figuras 22 e 23 respectivamente. Em ambos gráficos é notável a queda drástica de pacotes recebidos e o mesmo número de pacotes enviados em relação aos cenários sem alteração do TX e RX. Esse comportamento era esperado, já que tais taxas não influenciam na tentativa do nó de enviar o pacote, mas influenciam fortemente se o destinatário do nó conseguirá receber a mensagem. Neste caso a queda drástica se deu pelo fato de ter diminuído ao mesmo tempo tanto o sucesso de envio quanto o do receptor.

Figura 22: Histograma dos pacotes enviados e recebidos com os nós espalhados aleatoriamente pelo COOJA com TX e RX em 50%.

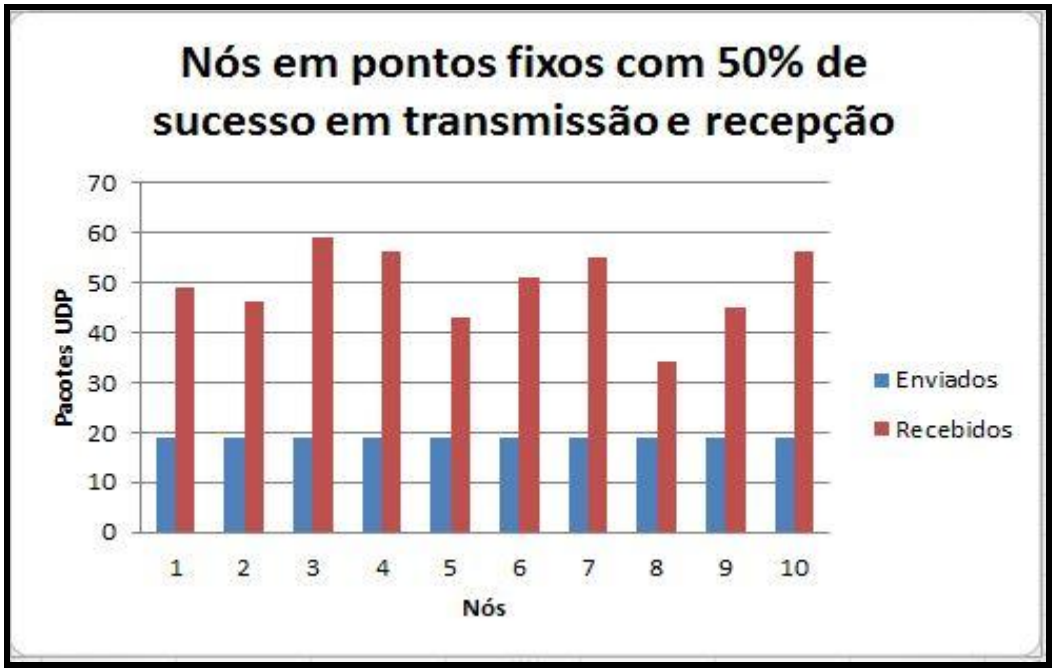

Figura 23: Histograma dos pacotes enviados e recebidos com os nós em pontos fixos e TX e RX em 50%.

5.3.3 API UDP:

Mesmo com um grau de dificuldade um pouco mais elevado, esse subtópico trata sobre a simulação da API UDP comum do Contiki. Para realizar esta simulação, primeiro buscaram-se as regras de compilação necessária para a execução da mesma, essas regras são (além daquelas necessárias para qualquer simulação, que foram apresentadas na figura 7):

- WITH_UIP6=1: Comando que ativa a utilização da pilha *uIP*, neste caso com IPv6.
- UIP_CONF_IPV6=1: É um identificador (*flag*) usado para ativar o IPv6, pois por *default* o Contiki utiliza o IPv4.
- UIP_CONF_IPV6_CHECKS=1: Também é um identificador, porém ele é o responsável por garantir que os pacotes que chegam ao nó estejam corretamente formados.
- UIP_STATISTICS=1: É o responsável por capturar as métricas dos pacotes recebidos, enviados e perdidos.

A figura 24 apresenta o *Makefile* da simulação.

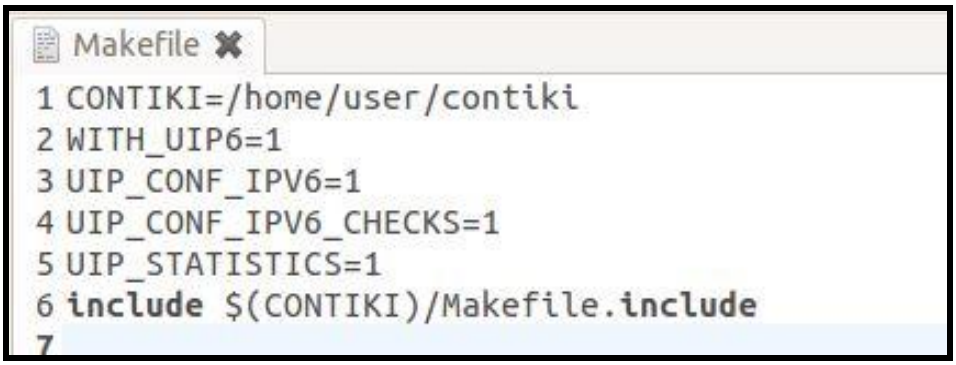

Figura 24: *Makefile* UDP.

Com o *Makefile* pronto buscou-se implementar o código de execução do nó. Primeiro foi adicionado as bibliotecas essenciais do Contiki: "contiki.h" e "contiki-net.h", depois a da saída padrão: <stdio.h> e por último a "cc2420.h" que inclui as funções para capturar os valores de piso RSSI e o RSSI do último pacote recebido.

Como esta simulação envia um *broadcast* para todos os nós utilizando a API UDP, primeiro deve-se utilizar a função udp_broadcast_new para criar uma nova conexão de *broadcast* UDP. Criada a conexão, somente a pilha *uIP* pode escolher o momento para enviar pacotes, porém a função tcpip_poll_udp força a pilha capturar a conexão especificada, permitindo assim, o envio de pacotes no momento desejado.

Nesta simulação, após a pilha escolher a conexão, usa-se a função uip_send para enviar um broadcast da *string* desejada. Após cada envio, imprimem-se as métricas da estrutura uip_stat, e apresenta os valores de RSSI com as funções:

- cc2420_rssi(): Leitura de piso RSSI;
- cc2420_last_rssi: RSSI do último pacote recebido.

Com todas as alterações devidamente concluídas e o código pronto, é mostrada a execução do mesmo na figura 25.

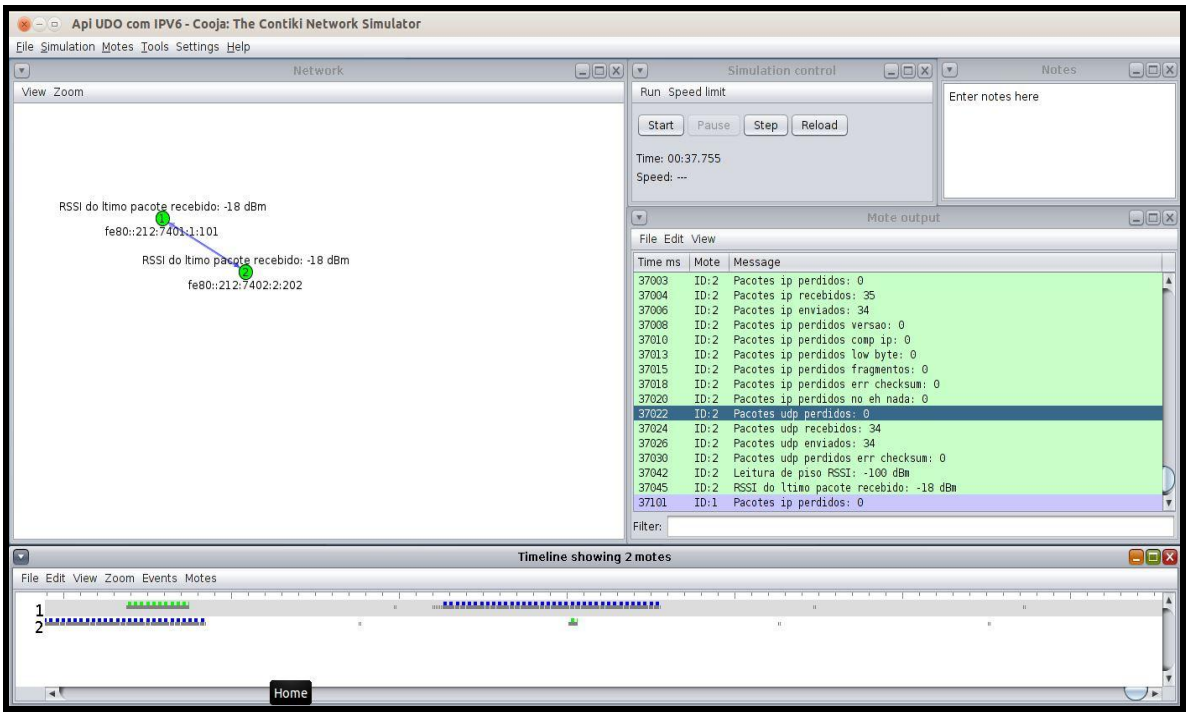

Figura 25: Simulação UDP.

- Resultados obtidos:

Por se tratar de análise de pacotes UDP, essa simulação segue os mesmos passos da simulação *simple* UDP, buscando analisar o comportamento de ambas em mesmas situações. O nó escolhido para a análise e apresentação de todas as métricas colhidas também foi o nó um.

➢ Relação nós – Contagem de pacotes: Realizou-se uma simulação com a disposição aleatória de dois nós e logo em seguida com dez nós, respeitando uma hora (tempo simulado) de cada simulação. Os valores obtidos estão apresentados nas tabelas 4, 5 e 6.

Tabela 4 - Relação do nó 1 com pacotes IP.

Nº nós	Perdidos	Recebidos	Enviados	Perdidos por erro de cabeçalho	Perdidos por erro no comprimento: *High byte*	Perdidos por erro no comprimento: Low byte	Perdidos devido a Fragmentação IP.	Perdidos Erro de checksum	Perdidos por não serem TCP, UDP ou ICMP
02	1	57	55	0	0	0	0	0	0
10	5	213	53	0	0	0	0	0	0

Tabela 5 - Relação do nó 1 com pacotes UDP.

Nº nós	Perdidos	Recebidos	Enviados	Perdidos Erro de checksum
02	0	56	55	0
10	0	208	53	0

Tabela 6 - RSSI obtidos pelo nó 1.

Nº nós	Base RSSI (dBm)	RSSI do último pacote (dBm)
02	-100	-37
10	-37	-37

A tabela 4 apresenta as métricas do nó 1 em relação à camada de rede, nota-se que por estar em uma disposição aleatória e com uma configuração em que o alcance da antena é de 50 metros, nem todos os nós estão conseguindo transmitir e receber pacotes dos demais. Esse fato é representado pelos valores de pacotes recebidos e enviados, pois em um ambiente com mais nós (10), o nó 1 acabou enviando menos pacotes do que quando estava na rede de dois nós. Essa diminuição ocorre devido aos nós fora do seu alcance e ainda, pela disputa daqueles nos quais se consegue transmitir, o que acaba aumentando o tráfego e consequentemente a perda de pacotes. Os outros valores referentes à perda de pacotes tratam situações causadas por erros de cabeçalhos, comprimento, fragmentação, *checksum* e por não serem pacotes TCP, UDP ou ICMP, os quais não tiveram ocorrências nesta simulação.

Já a tabela 5 relaciona o nó à camada UDP, esses valores mostram que todos os pacotes UDP produzidos na camada de transporte foram transmitidos pela camada de rede (igualdade de valores nas tabelas 4 e 5 em relação a pacotes enviados). Porém apresenta uma pequena divergência dos pacotes IP recebidos para os pacotes UDP recebidos no ambiente menor e uma maior diferença quando os dez nós estavam sendo simulados, fato este, que é explicado pelo número de pacotes IP perdidos e consequentemente deixaram de ser retransmitidos para a camada de transporte.

A tabela 6 foca nos valores de RSSI e mostra que a presença de mais nós não causou interferência na transmissão e recepção do nó 1, pois os valores de RSSI dos pacotes recebidos se manteve constante.

Saindo do âmbito de todas as métricas e focalizando nos pacotes UDP, simulou-se uma rede com disposição aleatória de nós que está apresentada na figura 26.

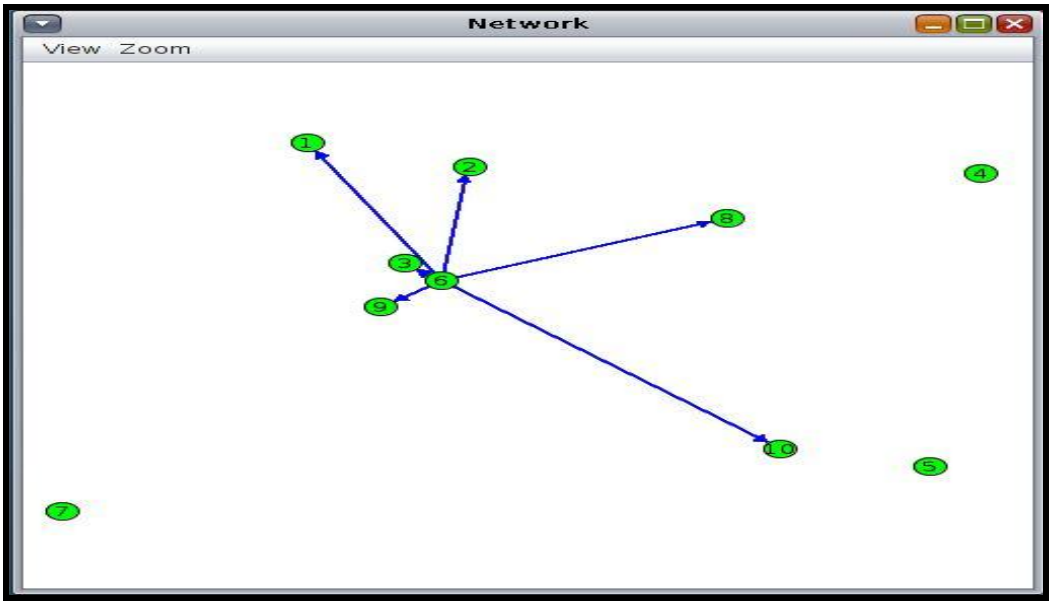

Figura 26: Nós dispostos aleatoriamente pelo COOJA.

Esse ambiente serviu para mostrar todos os pacotes UDP recebidos e enviados por todos os dez nós, tais valores estão expressos na figura 27. Note que esse gráfico confirma a relação da aleatoriedade dos nós com os pacotes recebidos, pois todos os dispositivos possuem uma linearidade na quantidade de pacotes enviados, porém uma variação em relação àqueles. Isso é explicado pelo fato dos nós 4, 5 e 7 (figura 26) estarem mais afastados e seus pacotes recebidos serem os menores entre os dos outros na figura 27.

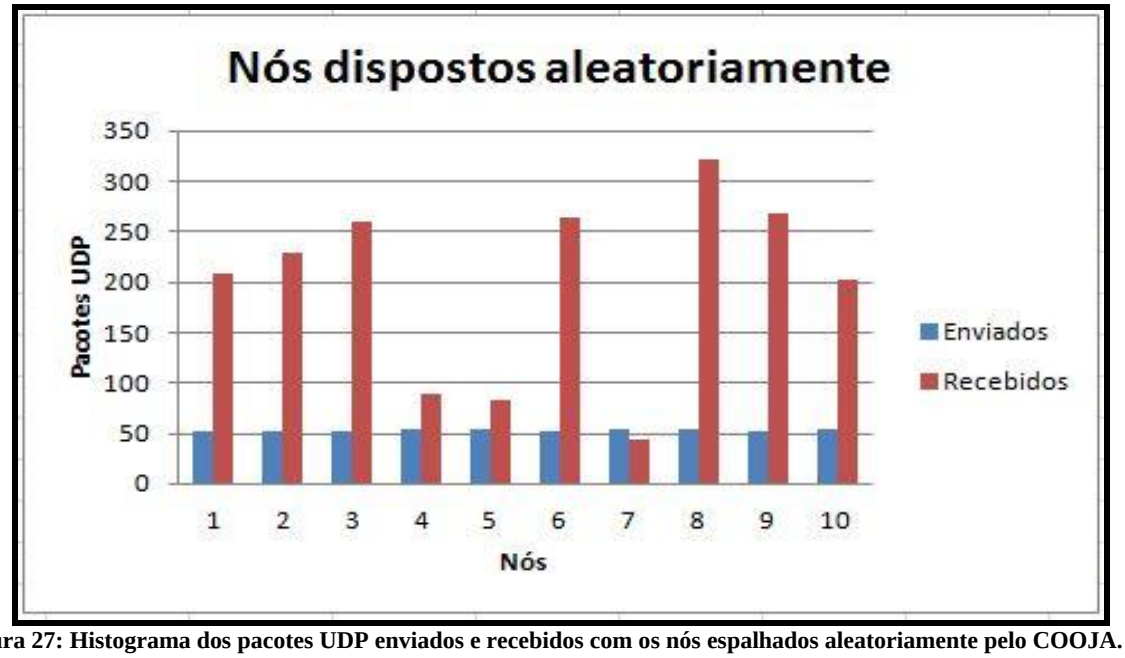

Figura 27: Histograma dos pacotes UDP enviados e recebidos com os nós espalhados aleatoriamente pelo COOJA.

Para tentar confrontar esses valores, montou-se o ambiente da figura 28, onde todos os nós estão na área de transmissão de todos os outros.

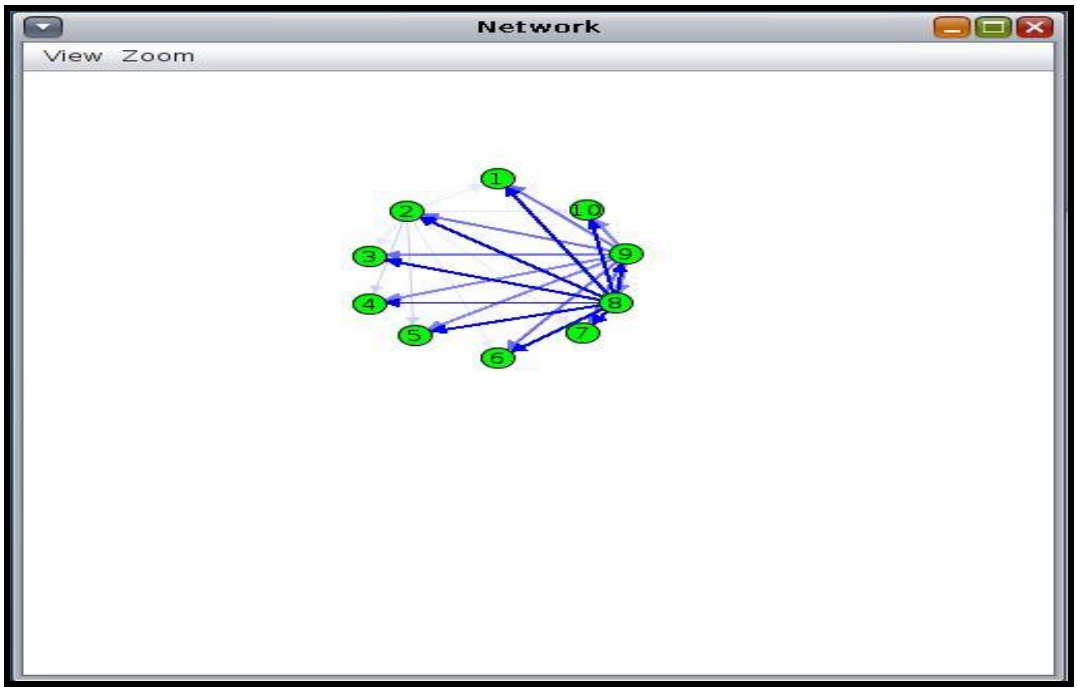

Figura 28: Nós dispostos em pontos fixos.

Com o ambiente pronto e após a simulação, gerou-se o gráfico da figura 29 com os valores obtidos. Através deste gráfico pode-se confirmar a dependência da disposição dos nós, pois note que existe uma estabilidade dos pacotes recebidos e enviados, mostrando que todos participaram de maneira igualitária nessa simulação de *broadcast*.

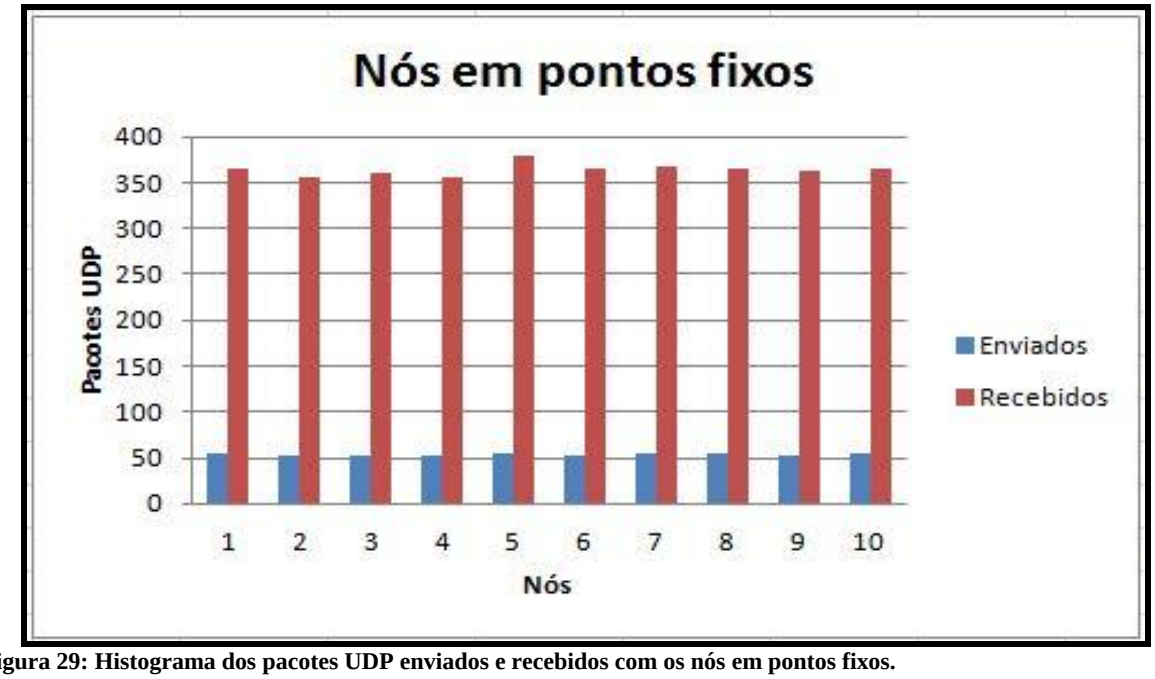

Figura 29: Histograma dos pacotes UDP enviados e recebidos com os nós em pontos fixos.

- Relação nós – TX (Taxa de sucesso de transmissão) e RX (Taxa de sucesso de recepção): Esta parte busca mostrar a relação existente entre a taxa de sucesso de TX e RX com os nós, simulando-os nos cenários com nós dispostos aleatoriamente e com nós dispostos em pontos fixos.

Seguindo o que foi padronizado, alterou-se o TX e RX para 50% no ambiente da figura 26 e foram obtidos os valores da figura 30. Note que novamente os pacotes enviados se mantêm estáveis e os recebidos instáveis, porém devido à redução da taxa de sucesso, ocorre também uma diminuição no número dos pacotes recebidos.

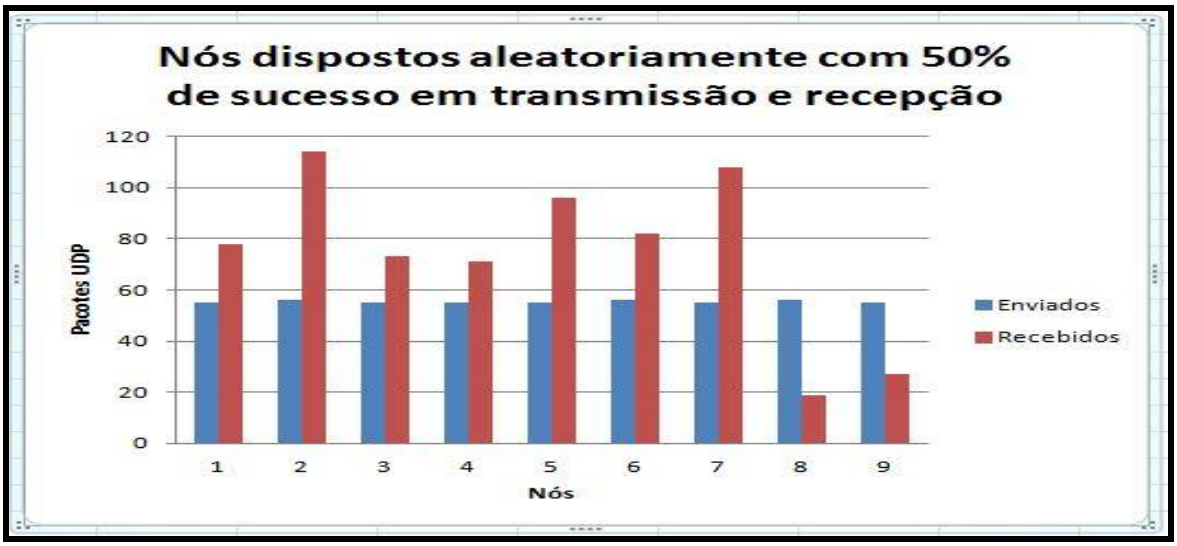

Figura 30: Histograma dos pacotes UDP enviados e recebidos com os nós espalhados aleatoriamente pelo COOJA com TX e RX em 50%.

Na mesma linha de pensamento, simulou-se o ambiente da figura 28 com a redução de 50% na taxa de sucesso. O resultado é mostrado na figura 31.

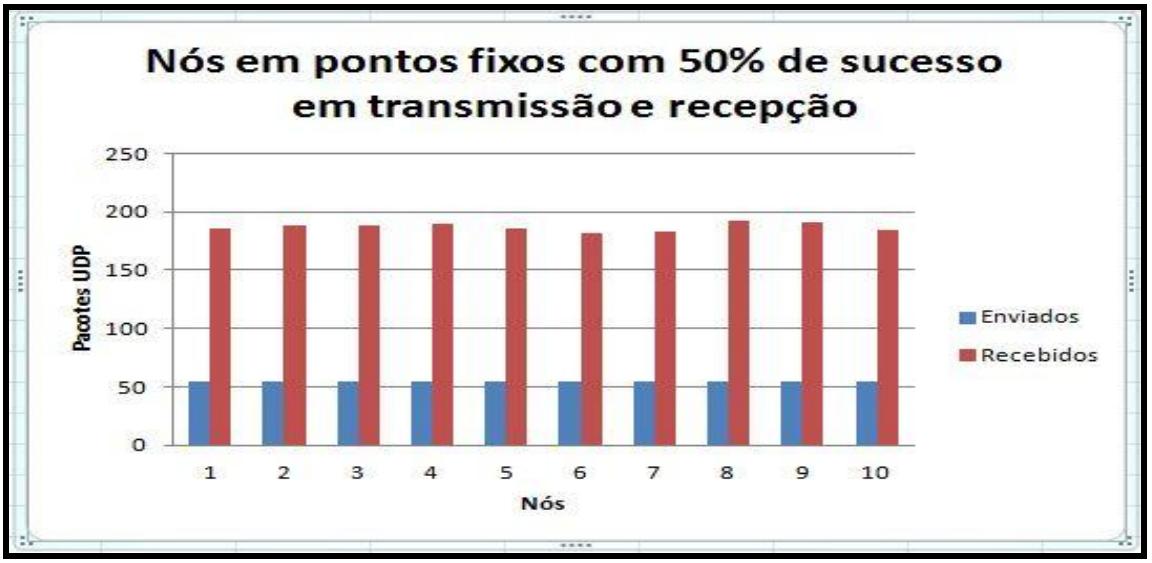

Figura 31: Histograma dos pacotes UDP enviados e recebidos com os nós em pontos fixos e TX e RX em 50%.

Como já era esperada, a simulação obteve valores semelhantes ao histograma da figura 29, com a diminuição na quantidade de pacotes recebidos devido à redução da taxa de sucesso.

5.3.4 TCP:

Nesta simulação é criado dois tipos de nós, o primeiro abre a conexão TCP com um IP específico através da função *tcp_connect* e envia intermitentemente uma mensagem para o endereço especificado, o qual foi denominado cliente. O segundo, chamado servidor, apenas escuta uma porta específica através da função *tcp_listen* e caso chegue um novo dado, emite um aviso que chegou dados no servidor. As estatísticas são retiradas da mesma forma já descrita na sessão de *simple UDP*.

Como o TCP exige uma abertura de conexão entre os nós, o endereço IPv6 é inserido manualmente no código do cliente e montado através da função *uip_ip6addr*, que transforma todos os campos do endereço em um único ponteiro. A montagem é necessária, pois a função *tcp_connect* descrita anteriormente aceita apenas um ponteiro.

Vários problemas foram encontrados nesse tipo de conexão e é preciso tomar alguns cuidados como, por exemplo, utilizar sempre a função *uip_htons* para conversão de bytes em ordem de host para ordem de rede, colocar cuidadosamente as macros que permitem espera de eventos TCP no código, observando a chegada dos *acknowledgments* e inserir o endereço do servidor no cliente de acordo com IP que o COOJA define seus nós. Este endereço pode ser visto assim que são adicionados nós na área de trabalho do simulador.

Outro ponto interessante é que sempre utilize as funções tcp_conect e tcp_listen, porque para as funções uip_conect e o uip_listen funcionar, precisa inserir a função uip_conn->appstate.p = PROCESS_CURRENT() que informa o atual estado da conexão para o estado atual do processo.

A figura 32 ilustra a simulação deste tipo de conexão com os nós devidamente endereçados e o *Makefile* não precisa ser alterado em relação à simulação do simples UDP mostrado na figura 16.

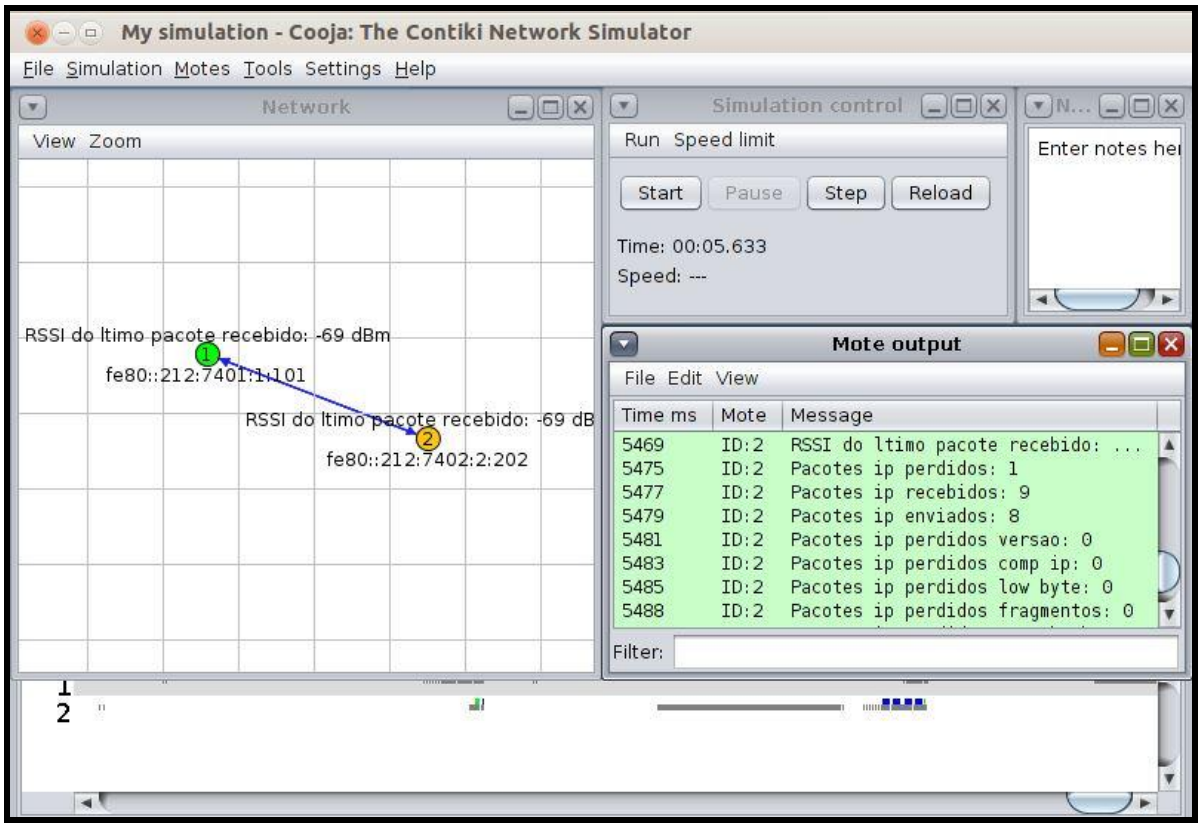

Figura 32: Simulação TCP.

- Resultados obtidos:

Na conexão TCP não é possível fazer um *broadcast* como nas demais simulações desse trabalho, então primeiramente foi feita uma simulação com dois nós, um cliente e um servidor, e em seguida, outra com cinco clientes e um servidor, sendo dispostos no espaço da área de trabalho do COOJA de forma que todos os nós conseguem alcançar o servidor.

Os resultados estão expostos na tabela 7 e 8, a primeira permite inferir que foi necessário o cliente retransmitir um pacote, já que houve um pacote perdido na camada de rede do servidor.

Na primeira simulação o cliente manteve total comunicação com o servidor durante toda a simulação, já que tínhamos apenas um cliente para um servidor, fato este que pode ser confirmado observando a tabela 7, a qual mostra claramente que todos os segmentos TCP enviados pelo cliente foram recebidos pelo servidor com exceção de um único segmento perdido na camada de rede pelo servidor, porém o TCP ativo no cliente retransmitiu o pacote.

Na segunda os clientes precisaram competir entre si o acesso ao servidor, fato que ocasionou grande perda de pacotes IP e grande quantidade de segmentos retransmitidos na camada de transporte. Essa competição foi tão intensa que os nós clientes de número 2 e 5 não

conseguiram acesso ao servidor em nenhum momento da simulação e consequentemente não foram obtidos resultados para os mesmos. Isso pode ser observado na tabela 8.

Tabela 7 - Estatísticas dos segmentos na conexão TCP entre um cliente e um servidor

	Cliente	Servidor
Segmentos TCP enviados	119	118
Segmentos TCP recebidos	118	118
Segmentos TCP retransmitidos	1	0
Segmentos TCP perdidos	0	0
Segmentos TCP perdidos por erro no cheksum	0	0
Segmentos TCP perdidos por erro de ACK	0	0
Segmentos SYNs TCP perdidos por poucas conexões disponíveis	0	0
Pacotes perdidos na camada de rede IP	0	1

Tabela 8 - Estatísticas dos segmentos na conexão TCP entre 5 clientes e 1 servidor

	Cliente 1	Cliente 2	Cliente 3	Cliente 4	Cliente 5	Servidor
Segmentos TCP enviados	20		24	15		39
Segmentos TCP recebidos	12		18	7		39
Segmentos TCP retransmitidos	14		9	12		0
Segmentos TCP perdidos	0		0	0		0
Segmentos TCP perdidos por	0		0	0		0

erro no cheksum						
Segmentos TCP perdidos por erro de ACK	0		0	0		0
Segmentos SYNs TCP perdidos por poucas conexões disponíveis	0		0	0		0
Pacotes perdidos na camada de rede IP	6		8	4		5

Ao diminuir a taxa de sucesso de transmissão e recepção para 50%, o cliente não conseguiu comunicação com servidor. Porém algumas alterações foram feitas visando à comunicação:

- Alterando RX até 100% e mantendo o TX em 50% não houve comunicação entre os nós.
- Mantendo RX em 100% e alterando TX para 60% não houve comunicação entre os nós.
- Mantendo RX em 100% e alterando TX para 70% não houve comunicação entre os nós.
- Mantendo RX em 100% e alterando TX para 80% houve comunicação e os resultados são exibidos na tabela 9. Nesta é notável a queda de pacotes enviados e recebidos se comparado com o mesmo ambiente sem alteração da taxa de sucesso de transmissão e recepção mostrada na tabela 7. A queda citada era esperada, pois com o TX em 80% a abertura de conexão entre os nós (necessária no protocolo TCP) ocorreu poucas vezes.
- Mantendo o TX em 100% e alterando o RX na faixa de 100% até 10% aberturas de conexões ocorreram normalmente.

Tabela 9 - Estatísticas dos segmentos na conexão TCP entre um cliente e um servidor com TX 80% e RX 100%.

	Cliente	Servidor
Segmentos TCP enviados	39	35
Segmentos TCP recebidos	35	35
Segmentos TCP retransmitidos	6	0
Segmentos TCP perdidos	0	0
Segmentos TCP perdidos por erro no cheksum	0	0
Segmentos TCP perdidos por erro de ACK	0	0
Segmentos SYNs TCP perdidos por poucas conexões disponíveis	0	0
Pacotes perdidos na camada de rede IP	0	0

5.3.5 RIME:

Neste trabalho simulou-se o módulo abc do Rime, que é um módulo que envia *broadcast*s anônimos. Por ser o módulo mais inferior da pilha, o *Makefile* não exige muitas regras para a sua compilação. A figura 33 mostra os comandos necessários para a simulação do mesmo.

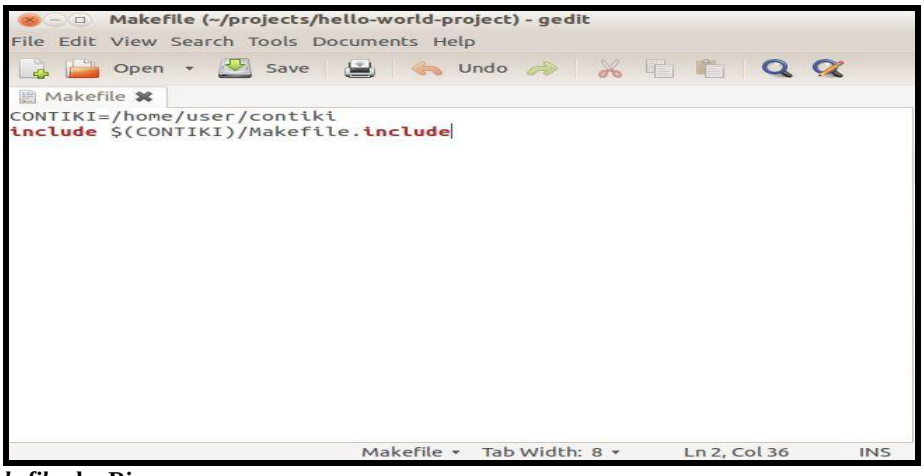

Figura 33: *Makefile* abc-Rime.

Seguindo para a implementação do nó, o módulo abc fornece quatro funções para o seu uso. Primeiro foi adicionado à biblioteca do Contiki: "contiki.h", seguida da biblioteca da pilha Rime:"net/rime.h" , depois a da saída padrão: <stdio.h> e por se tratar de uma pilha diferente, a biblioteca "rimestats.h" para incluir a estrutura que colhe as estatísticas da mesma e "print-stats.h" para importar a função print_stats(), a qual é responsável por imprimir os resultados da estatísticas.

Porém antes de partir para a implementação do nó, deve-se retirar a *flag* "#if RIMESTATS_CONF_ENABLED" e "#endif /* RIMESTATS_CONF_ENABLED */" de todos os arquivos "print-stats.c", e as definições "#define RIMESTATS_CONF_ON 1" e "#define RIMESTATS_CONF_ENABLED 1" do arquivo contiki-conf.h porque esses erros acabam interferindo na impressão das métricas obtidas.

Na implementação do nó, primeiro deve-se utilizar a função abc_open, para criar e configurar uma conexão abc. Neste momento deve passar como parâmetros, uma estrutura do tipo abc_conn, um canal no qual a conexão irá operar (escolher um número maior ou igual a 128, pois os canais menores que este número são reservados pelo sistema) e um ponteiro para uma função de *callback* que será chamada sempre que um pacote for recebido pelo canal.

Configurada a conexão, copia o pacote para o *buffer* e a função abc_send envia o pacote presente no *buffer* para a conexão escolhida. Após enviar o pacote, a função print_stats() apresenta as métricas obtidas. Porém vale ressaltar que se observou que os valores TX e RX, que correspondem a quantidade de pacotes transmitidos e recebidos respectivamente, não estão funcionando na função print_stats(). Ao efetuar uma busca sobre a causa dessa falha, notou-se que as funções RIMESTATS_ADD(TX) e RIMESTATS_ADD(RX) não estavam presente nos lugares necessários para realizar a contagem de tais métricas nesse módulo.

A figura 34 mostra a área de trabalho do simulador COOJA durante a execução desta simulação.

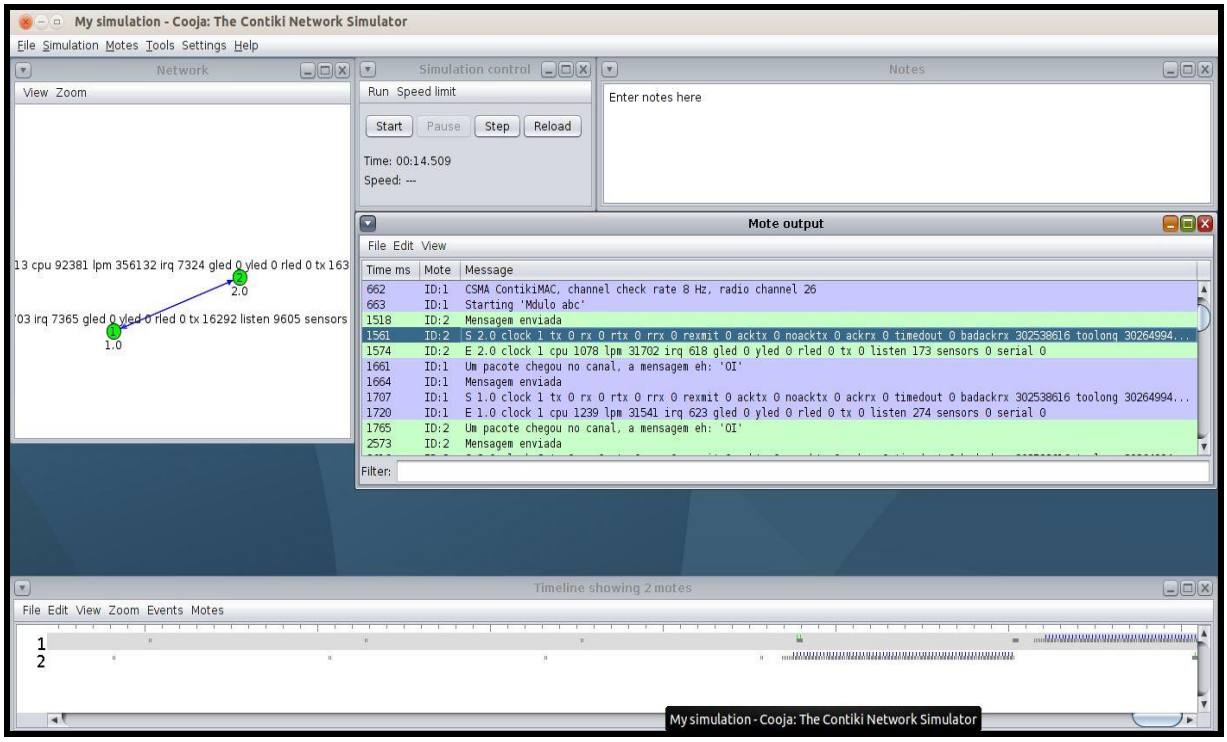

Figura 34: Simulação abc-Rime.

- Resultados obtidos:

Como foi relatada na subseção anterior, a simulação do módulo abc da pilha de comunicação Rime possui uma falha, e não captura os valores de pacotes enviados, transmitidos e perdidos, porém a função print_stats() fornece várias outras métricas.

Mesmo com essas dificuldades, simulou-se o módulo abc de Rime, adaptando as configurações escolhidas pra as medidas fornecidas pela pilha. Seguiu-se o mesmo tempo de simulação e a alteração no número de nós, porém não alterou a taxa de sucesso de transmissão por não conseguir captar os pacotes recebidos e enviados devido à falha do nó. Os valores obtidos pelo nó 1 estão mostrados nas tabelas 10 e 11.

Tabela 10 - Estatísticas da estrutura rimestats obtida pelo nó 1

Ambiente com o nº de nós	02	10
Endereço	1.0	1.0
Clock_seconds	59	59
TX	0	0
RX	0	0
Reliabletx	0	0
Reliablerx	0	0
Rexmit	0	0
Acktx	0	0
Noacktx	0	0
Ackrx	0	0
Timedout	0	0
Badackrx	302538616	302538616
Toolong	302649940	302649940
Tooshort	1596854792	1596854792
Badsynch	302256724	302256724
Badcrc	294412942	294412942
Contentiondrop	307512422	307512422
Sendingdrop	632038829	632038829

Tabela 11 - Estatísticas de energia utilizadas pelo sistema do nó 1

Ambiente com o nº de nós	**02**	**10**
Endereço	1.0	1.0
Clock_seconds	59	59
CPU	425007	505656
LPM	1520563	1450551
IRQ	31554	35611
LED_GREEN	0	0
LED_YELLOW	0	0
LED_RED	0	0
TRANSMIT	74679	62515
LISTEN	27061	191402
SENSORS	0	0
SERIAL	0	0

Note que a falha do módulo abc está representada na tabela 10 através das estatísticas referentes à transmissão e recepção de pacotes cujos valores são zero. Fica claro também que os valores corelacionados, como perda de pacotes por um *ack* mal recebido (*badackrx*) até a linha *sendingdrop*, que se refere aos pacotes perdidos quando o módulo está enviando pacotes, está mostrando valores que se assemelham a "pontos flutuantes" devido ao cálculo com valores inexistentes (TX e RX).

Já a tabela 11 mostra as estatísticas de consumo de energia do nó virtual, note que esses valores condizem com a realidade, pois na simulação de dez nós obteve-se um tempo de consumo de CPU maior do que na simulação de dois nós, ou seja, a CPU ficou mais tempo ligada.

Vale resaltar a importância da tabela 11, por mostrar o tempo de consumo de energia de todos os componentes do nó (CPU, LPM, IRQ, *LED's*, consumo de transmissão, consumo enquanto a antena está ouvindo, consumo dos sensores e consumo da porta serial), o que permite calcular o tempo de vida, pois no âmbito da *IoT*, o ciclo de vida dos nós estará relacionado com o consumo da fonte de alimentação do sistema.

5.3.6 REST

Neste trabalho, tanto o código do *Makefile* quanto o código do nó, foram adaptados do exemplo REST de Adam Dunkels (disponível no Contiki). Foram realizadas as alterações necessárias para capturar as métricas do sistema.

- O primeiro ponto observado foi o arquivo "project-conf.h", o qual definia todas as configurações do projeto. Após a sua analise foi resolvido por deixá-lo intacto. Feito isto, seguiu-se para o arquivo de compilação, e acrescentaram as regras WITH_UIP6=1 e UIP_STATISTICS=1 às regras já existentes no *Makefile*. A figura 35 apresenta o arquivo resultante.

Makefile

```
all: rest-server-example coap-client-example

ifndef TARGET
TARGET=sky
endif

CONTIKI=/home/user/contiki

WITH_UIP6=1
UIP_CONF_IPV6=1
UIP_STATISTICS=1

WITH_COAP = 1

CFLAGS += -DPROJECT_CONF_H=\"project-conf.h\"

ifeq ($(WITH_COAP), 1)
CFLAGS += -DWITH_COAP
APPS += rest-coap
else
CFLAGS += -DWITH_HTTP
APPS += rest-http
endif

include $(CONTIKI)/Makefile.include

$(CONTIKI)/tools/tunslip6:      $(CONTIKI)/tools/tunslip6.c
        (cd $(CONTIKI)/tools && $(MAKE) tunslip6)

connect-router: $(CONTIKI)/tools/tunslip6
        sudo $(CONTIKI)/tools/tunslip6 aaaa::1/64

connect-router-cooja:   $(CONTIKI)/tools/tunslip6
        sudo $(CONTIKI)/tools/tunslip6 -a 127.0.0.1 aaaa::1/64
```

Figura 35: *Makefile* REST.

Em relação ao código de execução do nó, Adam Dunkels produziu um nó cliente COAP (*Constrained Application Protocol*) que realiza requisições ao nó servidor REST, que responde dependendo da informação ou recurso presente na requisição. Está simulação alterou o código do cliente para manter as mesmas funcionalidades anteriores, porém com o

acréscimo da contagem dos pacotes IP, UDP e dos valores de RSSI, através das mesmas estruturas e funções utilizadas nas simulações anteriores da pilha *uIP*.

A figura 36 mostra a área de trabalho do simulador COOJA durante a execução desta simulação.

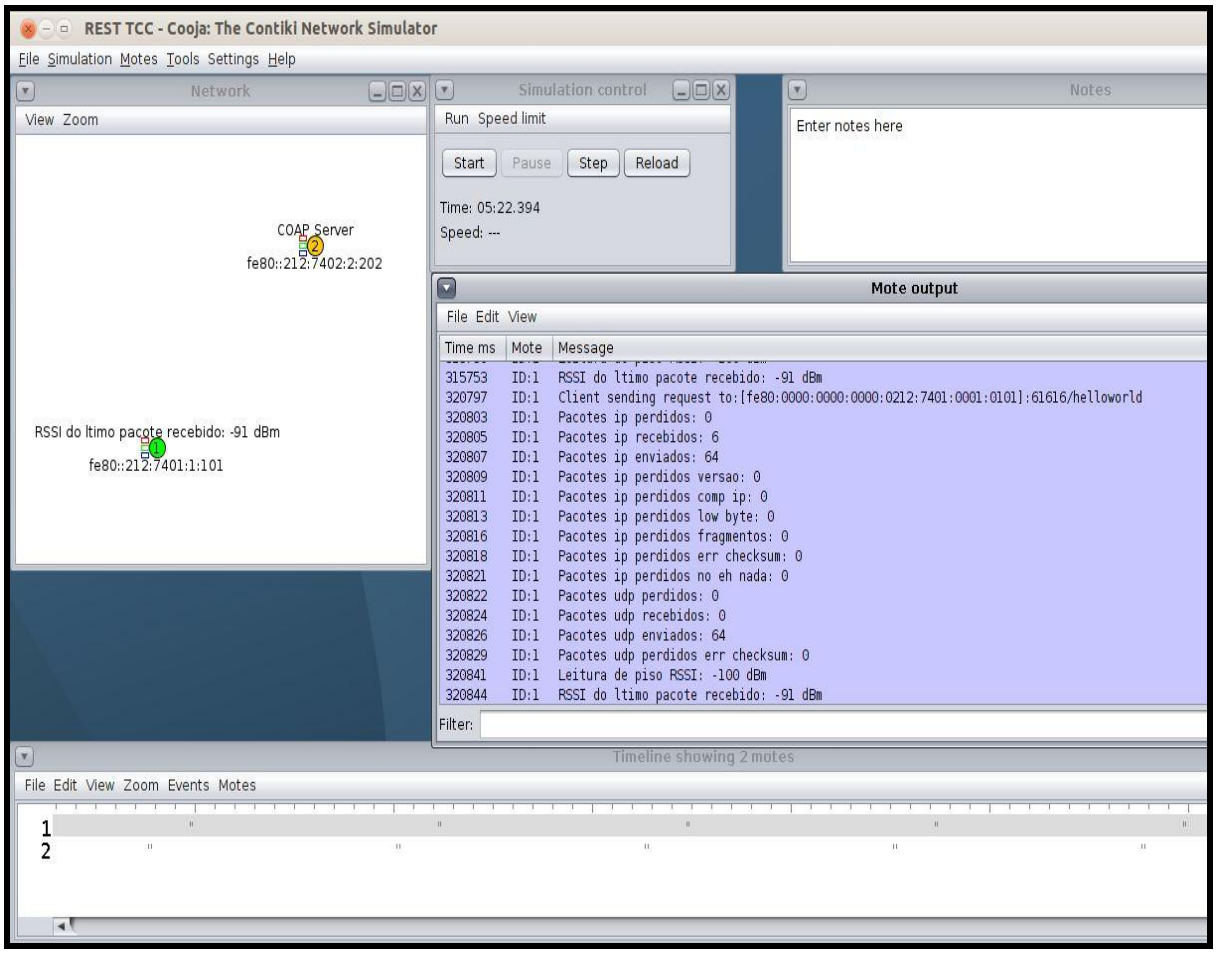

Figura 36: Simulação REST.

- Resultados obtidos:

Com a simulação UDP, escolheu-se o nó servidor para ser analisado e seguiu-se todas as especificações impostas. Porém a única diferença é que foi escolhido nove nós cliente e um único servidor.

- Relação nós – Contagem de pacotes: Realizou-se a simulação de um ambiente fixo com o servidor no centro e os clientes ao redor, e obtiveram-se os valores apresentados nas tabelas 12, 13 e 14.

Tabela 12 - Relação nó servidor com pacotes IP

Nº nós	Perdidos	Recebidos	Enviados	Perdidos por erro de cabeçalho	Perdidos por erro no comprimento: *High byte*	Perdidos por erro no comprimento: Low byte	Perdidos devido a Fragmentação IP.	Perdidos Erro de checksum	Perdidos por não serem TCP, UDP ou ICMP
02	0	1	11	0	0	0	0	0	0
10	0	14	12	0	0	0	0	0	0

Tabela 13 - Relação nó servidor com pacotes UDP

Nº nós	Perdidos	Recebidos	Enviados	Perdidos Erro de checksum
02	0	0	11	0
10	0	10	11	0

Tabela 14 - RSSI obtido nó servidor

Nº nós	Piso RSSI (dBm)	RSSI do último pacote (dBm)
02	-100	-91
10	-95	-80

Pela análise dos resultados, esse sistema comprovou a importância da qualidade do sinal para a comunicação. Isto fica evidente pelos baixos valores RSSI e pela pouca quantidade de pacotes IP e UDP recebidos e enviados na simulação.

- Relação nós – Taxa de sucesso de transmissão: Nesta parte, foi alterada a taxa de sucesso de transmissão para observar o efeito em um sistema cliente-servidor em um ambiente fixo com o servidor no centro e os clientes ao redor. Os valores obtidos estão apresentados nas tabelas 15, 16 e 17.

Tabela 15 – Relação nó servidor com pacotes IP

Nº nós	Perdidos	Recebidos	Enviados	Perdidos por erro de cabeçalho	Perdidos por erro no comprimento: *High byte*	Perdidos por erro no comprimento: Low byte	Perdidos devido a Fragmentação IP.	Perdidos Erro de checksum	Perdidos por não serem TCP, UDP ou ICMP
02	0	1	13	0	0	0	0	0	0
10	0	8	15	0	0	0	0	0	0

Tabela 16 - Relação nó servidor com pacotes UDP

Nº nós	Perdidos	Recebidos	Enviados	Perdidos Erro de checksum
02	0	0	13	0
10	0	3	11	0

Tabela 17 - RSSI obtido nó servidor.

Nº nós	Base RSSI (dBm)	RSSI do último pacote (dBm)
02	-100	-91
10	-100	-80

Nesta simulação, diminuir a taxa de sucesso não alterou a qualidade do sinal, podendo assimilar que a própria estrutura da simulação, a qual só possuía um servidor para nove clientes, acabou interferindo no envio e na recepção de pacotes.

Vale ressaltar que durante todas as simulações, esse servidor REST se comportou de maneira inesperada, pois somente em poucas vezes, o mesmo conseguiu responder as solicitações do cliente. O motivo dessa instabilidade no servidor não foi descoberta neste trabalho.

6 CONCLUSÃO

As simulações descritas neste trabalho apresentaram resultados esperados conforme as alterações eram feitas em número de nós e disposição dos mesmos na área de trabalho do COOJA, com exceção de REST que se comportou de forma inesperada em todas as simulações como descrito anteriormente.

Apesar das dificuldades encontradas neste trabalho como, por exemplo, precisar modificar as bibliotecas do Contiki para o uso da função uip_stats ou a sincronização necessária entre cliente e servidor nos casos de conexões TCP, os resultados obtidos mostram que tanto a pilha de comunicação *uIP* quanto a pilha de comunicação Rime funcionam, e provavelmente também funcionem no *Tmote sky* ou em qualquer outra plataforma compatível com o Contiki, desde que sejam feitas as devidas alterações de compatibilidade.

Algumas das estatísticas não foram obtidas e outras ficaram sempre em zero como os pacotes perdidos TCP e UDP. Acredita-se que estes não saíram do zero devido ao fato dos pacotes perdidos contarem como pacotes IP perdidos já que os pacotes IP encapsulam os pacotes TCP e UDP, os mesmos não são contados. Porém não se pode afirmar que este é o motivo real da situação descrita ter ocorrido, podendo o motivo ser devido a outros fatores desconhecidos pelos autores do presente trabalho.

O objetivo deste trabalho foi simular a leve pilha de protocolos *uIP* utilizando da versão 6 (IPV6) com os protocolos UDP e TCP com o intuito de mostrar que é possível implementar em nós reais a Internet das coisas, sendo assim, vários trabalhos futuros podem ser feitos a partir deste, como por exemplo:

1. Corrigir o módulo abc para captar os valores de TX e RX;
2. Explorar melhor a pilha Rime, realizando experimentos com outros módulos;
3. Tentar obter a estatística de energia da pilha de comunicação *uIP*;
4. Executar os códigos deste trabalho em uma plataforma real do *Tmote sky*;
5. Expandir a comunicação TCP com ipv6 para uso de *protosockets*, o qual se tentou com utilização do IPV6, mas só funcionou com IPV4, talvez devido à sincronização necessária para comunicação;
6. Modificar o ambiente do COOJA para que haja algum tipo de nó perdido na camada de transporte (TCP ou UDP) e não somente na camada de rede IP, além das demais estatísticas que não saíram do zero. Lembrando a possibilidade da função uip_stats não estar funcionando nos casos citados;

7. Ampliar a comunicação TCP para vários servidores e clientes se comunicando automaticamente, sem a necessidade de fixar o endereço ipv6 do servidor no cliente;

8. Simular os códigos apresentados neste trabalho ou utilizá-los com base para simulações no ambiente *minimal net*.

REFERENCIAS

ASHTON, K. **That 'Internet of Things' Thing. Online RFID Journal.** Publicado em 2009. Disponível em: <http://www.itrco.jp/libraries/RFIDjournal-That%20Internet%20of%20Things%20Thing.pdf>. Acessado em 23 de setembro de 2013.

ATZORI L.; IERA A.; MORABITO, G. **The Internet of Things: A survey.** Computer Networks Science Direct. 2010.

CONTIKI 2.6. **Multi-hop reliable bulk data transfer** publicado em 2012. Disponível em: <http://contiki.sourceforge.net/docs/2.6/a01736.html>. Acessado em 24 de setembro de 2013.

CONTIKI. **The uIP TCP/IP stack.** Publicado em 2012. Disponível em: <http://contiki.sourceforge.net/docs/2.6/a01793.html>. Acessado em 08 de abril de 2014.

DEERING, S.; HINDEN, R. **Internet Protocol**. Version 6 (IPv6) Specification, RFC Editor, 1998.

DUNKELS, A. **Full TCP/IP for 8-bit architectures.** Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS 2003) (May 2003).

DUNKELS, A. **Rime - a lightweight layered communication stack for sensor networks.** European Conference on Wireless Sensor Networks (EWSN), January 2007, Delft, The Netherlands. Disponível em: <http://www.sics.se/~adam/dunkels07rime.pdf>. Acessado em 06 de abril de 2014.

DUNKELS, A.; GR¨ONVALL, B.; VOIGT, T. **Contiki – a lightweight and _exible operating system for tiny networked sensors.** Proceedings of the First IEEE Workshop on Embedded Networked Sensors, Tampa, Florida, USA, Nov. 2004.

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures.** Doctoral dissertation, University of California, Irvine. Publicado em 2000. Disponível em: <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>. Acessado em: 10 de abril de 2014.

FILHO, S. C. **Endereçamento IP** publicado em 2003**.** Disponível em: <http://www.cpdee.ufmg.br/~seixas/PaginaSDA/Download/DownloadFiles/Endere%E7amentoIP.pdf>. Acessado em 26 de setembro de 2013.

GIUSTO, D.; IERA, A.; MORABITO, G.; ATZORI, L. (Eds.). **The Internet of Things.** Springer, 2010. ISBN: 978-1-4419-1673-0.

GNU MAKE. Free Software Foundation. Disponível em: <https://www.gnu.org/software/make/>. Acessado em: 10 de abril de 2014.

INTERNET PROTOCOL. **RFC 791, Internet Engineering Task Force**. September 1981.

KIRSCHE, M. **Simulating the Internet of Things in a Hybrid Way**. Proceedings of the Networked Systems (NetSys) 2013 PhD Forum. Publicado em março de 2013. Disponível em: <https://www-rnks.informatik.tu cottbus.de/content/unrestricted/staff/mk/Publications/NetSys_2013-PhD_ForumKirsche.pdf>. Acessado em 24 de setembro de 2013.

KLAUCK, R.; KIRSCHE, M. **A Case Study of a DNS-based Discovery Service for the Internet of Things.** Publicado em 2012. Disponível em: <https://www-rnks.informatik.tu-cottbus.de/content/unrestricted/staff/mk/Publications/AdHocNow_2012Klauck_Kirsche.pdf>. Acessado em 16 de setembro de 2013.

KUROSE, James F.; ROSS, Keith W. **Redes de computadores e a Internet: uma abordagem top-down.** Tradução Opportunity translations; revisão técnica Wagner Zucchi. – 5. Ed. – São Paulo : Addison Wesley, 2010.

LEVIS, P.; LEE, N.; WELSH, M.; CULLER, D. **Tossim: accurate and scalable simulation of entire tiny os applications.** Proceedings of the _rst international conference on Embedded networked sensor systems, pages 126.137, 2003.

MIYAGI, P. E. **Introdução a Simulação Discreta.** Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos, SP, p.1-12. 2006. Disponível em: <http://minerva.ufpel.edu.br/~alejandro.martins/dis/2012_2/simulacao/Apostila_Simulacao.pdf>. Acessado em 08 de abril de 2014.

ÖSTERLIND, F.; DUNKELS, A.; ERIKSSON, J.; FINNE, N.; VOIGT, T. **Cross-Level Sensor Network Simulation with Cooja.** Proceedings of the First IEEE International Workshop on Practical Issues in Building Sensor Network Applications. (SenseApp 2006), Tampa, FL, USA, 14 November 2006.

PARAMESWARAN, A. T.; MOHAMMAD, I. H; UPADHYAYA, S. **Is RSSI a Reliable Parameter in Sensor Localization Algorithms – An Experimental Study.** 2009. Disponível em: <https://www.cse.buffalo.edu/srds2009/F2DA/f2da09_RSSI_Parameswaran.pdf>. Acessado em 08 de abril de 2014.

PETERSON, L.L; DAVIE, S. B. **Computer Networks: A System Approach.** 2003. Third Edition: A Systems Approach, 3rd Edition.

POSTEL, J. **Internet protocol.** RFC 791, Internet Engineering Task Force, September 1981.

POSTEL, J. **Transmission control protocol.** RFC 793, Internet Engineering Task Force, September 1981.

SANTOS, Mauricio Pereira dos. **Introdução à simulação Discreta.** IME. 1999 .Disponível em: <http://www.mpsantos.com.br/simul/arquivos/simul.pdf>. Acessado em 08 de abril de 2014.

SANTUCCI, G. **The Internet of Things: Between the Revolution of the Internet and the Metamorphosis of Objects**. Publicado em 2010. Disponível em: <http://cordis.europa.eu/fp7/ict/enet/documents/publications/iot-between-the-internet-revolution.pdf>. Acessado em 06 de setembro de 2013.

SRIPANIDKULCHAI, K.; MAGGS, B.; ZHANG, H. **"An analysis of live streaming workloads on the Internet".** Proceedings 4th ACM SIGCOMM Internet Measurement Conference, Taormina, p. 41-54, 2004. Disponível em: < http://conferences.sigcomm.org/imc/2004/papers/p41-sripanidkulchai.pdf>. Acessado em 06 de abril de 2014.

TMOTE SKY. **Ultra low power IEEE 802.15.4 compliant wireless sensor module.** Datasheet publicado em 2006. MoteiV. Disponível em: < http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/tmote-sky-datasheet.pdf>. Acessado em 23 de setembro de 2013.

TSIFTES, N.; ERIKSSON, J.; DUNKELS A. **Low-power wireless IPv6 routing with ContikiRPL.** Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks. April 12-16, 2010, Stockholm, Sweden [doi>10.1145/1791212.1791277].

UCKELMANN, D.; HARRISON, M.; MICHAHELLES, F. **Architecting the Internet of Things.** Publicado em 2009. Disponível em: <http://www.cui-zy.cn/Recommended/Linux/Architecting_the_Internet_of_Things.pdf>. Acessado em 23 de setembro de 2013.

VARGA, A.; HORNING, R. **"An Overview of the OMNeT++ Simulation Environment."** Proceedings of the First Conference on Simulation Tools and Techniques for Communications, Networks and Systems. (Simutools 2008). ICST, 2008, PP. 1-10.

ZIGBEE-READY RF TRANSCEIVER. Disponível em: <http://pdf1.alldatasheet.com/datasheet-pdf/view/454119/TI/CC2420.html>. Acessado em 08 de abril de 2014.

ANEXO A – *Makefile* da simulação *Hello World*

```
CONTIKI=/home/user/contiki
include $(CONTIKI)/Makefile.include
```

ANEXO B – Código da simulação *Hello World*

```
/*
 * \file
 *         Hello World Project
 * \author
 *
 *           Diego Assis Siqueira Gois - diego.se.ita@gmail.com
 *           João Paulo Andrade Lima - dm.joaopaulo@gmail.com
 */

#include "contiki.h"
#include "dev/button-sensor.h"
#include "dev/leds.h"
#include <stdio.h>
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
PROCESS_THREAD(hello_world_process, ev, data){
        PROCESS_BEGIN();
        SENSORS_ACTIVATE(button_sensor);
        leds_off(LEDS_GREEN);
        leds_on(LEDS_RED);
        printf("Hello, world\n");
        while(1){
                static uint32_t seconds = 5;
                static struct etimer et; // Define the timer
                PROCESS_WAIT_EVENT();
                if (ev == sensors_event && data == &button_sensor){
                        printf("Button Pressed!\n");
```

```
                etimer_set(&et, CLOCK_SECOND*seconds);
                printf("+ Timer started +\n");
                }
        if(etimer_expired(&et)) {
        leds_toggle(LEDS_GREEN);
                leds_toggle(LEDS_RED);
                etimer_reset(&et);
                printf("+ Timer finished +\n");
                }
        }
PROCESS_END(); }
```

ANEXO C – *Makefile* da simulação *Simple* UDP

```
CONTIKI=/home/user/contiki/
WITH_UIP=1
UIP_CONF_IPV6=1
UIP_STATISTICS=1
UIP_CONF_IPV6_CHECKS=1
include $(CONTIKI)/Makefile.include
```

ANEXO D – Código da simulação *Simple* UDP

```
/*
 * \file
 *       Simple UDP project
 * \author
 *
 *          Diego Assis Siqueira Gois - diego.se.ita@gmail.com
 *          João Paulo Andrade Lima - dm.joaopaulo@gmail.com
 *
 * O programa abre uma conexão broadcast UDP e envia um pacote
 * a cada 3 segundos.
 */
#include "net/uip-icmp6.h"
#include "net/uip-nd6.h"
#include "cc2420.h"
#include "contiki.h"
#include "net/uip.h"
#include "net/uip-ds6.h"
#include "simple-udp.h"

static struct simple_udp_connection broadcast_connection;
PROCESS(example_program_process, "Example process");
AUTOSTART_PROCESSES(&example_program_process);
static struct etimer timer;
```

```
extern struct uip_stats uip_stat;
static int rssi;
static char rssi2;
static void
receiver(struct simple_udp_connection *c,
        const uip_ipaddr_t *sender_addr,
        uint16_t sender_port,
        const uip_ipaddr_t *receiver_addr,
        uint16_t receiver_port,
        const uint8_t *data,
        uint16_t datalen)
{

          //printf("Data received on port %d from port %d with length %d Date: %s\n",
       // receiver_port, sender_port, datalen, data);

}

PROCESS_THREAD(example_program_process, ev, data)
{
          uip_init();
          uip_ipaddr_t addr;

          PROCESS_BEGIN();
          char line_buffer[80];

          simple_udp_register(&broadcast_connection, 1234,NULL, 1234,receiver);

          while(1) {
          etimer_set(&timer, 3*CLOCK_SECOND);
          PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&timer));

          uip_create_linklocal_allnodes_mcast(&addr);
          simple_udp_sendto(&broadcast_connection,"Teste", 5 , &addr);
```

```
                printf ("Pacotes ip perdidos: %d\n",uip_stat.ip.drop);
                printf ("Pacotes ip recebidos: %d\n",uip_stat.ip.recv);
                printf ("Pacotes ip enviados: %d\n",uip_stat.ip.sent);
                printf ("Pacotes ip perdidos versao: %d\n",uip_stat.ip.vhlerr);
                printf ("Pacotes ip perdidos comp ip: %d\n",uip_stat.ip.hblenerr);
                printf ("Pacotes ip perdidos low byte: %d\n",uip_stat.ip.lblenerr);
                printf ("Pacotes ip perdidos fragmentos: %d\n",uip_stat.ip.fragerr);
                printf ("Pacotes ip perdidos err checksum: %d\n",uip_stat.ip.chkerr);
                printf ("Pacotes ip perdidos não eh nada: %d\n",uip_stat.ip.protoerr);

                printf ("Pacotes udp perdidos: %d\n",uip_stat.udp.drop);
                printf ("Pacotes udp recebidos: %d\n",uip_stat.udp.recv);
                printf ("Pacotes udp enviados: %d\n",uip_stat.udp.sent);
                printf ("Pacotes udp perdidos err checksum: %d\n",uip_stat.udp.chkerr);
                //Obs: EStudo comprovou que o rssi não pode ser usado para avaliar a distância
do nós
                rssi = cc2420_rssi(); // Leitura de piso RSSI
                rssi2 = cc2420_last_rssi; // RSSI do último pacote recebido
                printf("Leitura de piso RSSI: %d dBm\n",rssi-45); // Esse 45 é o valor de offset
                printf("RSSI do ultimo pacote recebido: %d dBm\n",rssi2-45);

  }

  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

ANEXO E – *Makefile* da simulação da API UDP

```
CONTIKI=/home/user/contiki
WITH_UIP6=1
UIP_CONF_IPV6=1
UIP_CONF_IPV6_CHECKS=1
UIP_STATISTICS=1
include $(CONTIKI)/Makefile.include
```

ANEXO F – Código da simulação da API UDP

```
/*
 * \file
 *        API UDP
 * \author
 *          João Paulo Andrade Lima - dm.joaopaulo@gmail.com
 *          Diego Assis Siqueira Gois - diego.se.ita@gmail.com
 */

#include "contiki.h"
#include "contiki-net.h"
#include "cc2420.h" //USAR RSSI
#include <stdio.h>


PROCESS(example_program_process, "UDP Broadcast com IPV6");
AUTOSTART_PROCESSES(&example_program_process);

static struct uip_udp_conn *c;
static struct etimer timer;
extern struct uip_stats uip_stat;
int rssi;
char rssi2;
PROCESS_THREAD(example_program_process, ev, data)
```

```
{
 PROCESS_BEGIN();
        uip_init();
        c = udp_broadcast_new(UIP_HTONS(4321), NULL);

        while(1){
                etimer_set(&timer, CLOCK_SECOND);
                PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&timer));

                tcpip_poll_udp(c);
                PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
                uip_send("Hello", 5);
                printf ("Pacotes ip perdidos: %d\n",uip_stat.ip.drop);
                printf ("Pacotes ip recebidos: %d\n",uip_stat.ip.recv);
                printf ("Pacotes ip enviados: %d\n",uip_stat.ip.sent);
                printf ("Pacotes ip perdidos versao: %d\n",uip_stat.ip.vhlerr);
                printf ("Pacotes ip perdidos high byte: %d\n",uip_stat.ip.hblenerr);
                printf ("Pacotes ip perdidos low byte: %d\n",uip_stat.ip.lblenerr);
                printf ("Pacotes ip perdidos fragmentos: %d\n",uip_stat.ip.fragerr);
                printf ("Pacotes ip perdidos err checksum: %d\n",uip_stat.ip.chkerr);
                printf ("Pacotes ip perdidos que não eh TCP, UDP, ICMP:
%d\n",uip_stat.ip.protoerr);

                printf ("Pacotes udp perdidos: %d\n",uip_stat.udp.drop);
                printf ("Pacotes udp recebidos: %d\n",uip_stat.udp.recv);
                printf ("Pacotes udp enviados: %d\n",uip_stat.udp.sent);
                printf ("Pacotes udp perdidos err checksum: %d\n",uip_stat.udp.chkerr);

                rssi = cc2420_rssi(); // Leitura de piso RSSI
                rssi2 = cc2420_last_rssi; // RSSI do último pacote recebido
                printf("Leitura de piso RSSI: %d dBm\n",rssi-45); // Esse 45 é o valor de offset
                printf("RSSI do último pacote recebido: %d dBm\n",rssi2-45);
        }
 PROCESS_END();
```

```
}
/*---------------------------------------------------------------------------*/
```

ANEXO G – *Makefile* da simulação do módulo abc de Rime

```
CONTIKI=/home/user/contiki
include $(CONTIKI)/Makefile.include
```

ANEXO H – Código da simulação do módulo abc de Rime

```
/*
 * \file
 *         Módulo abc em Rime
 * \author
 *              João Paulo Andrade Lima - dm.joaopaulo@gmail.com
 *               Diego Assis Siqueira Gois - diego.se.ita@gmail.com
 */

#include "contiki.h"
#include "net/rime.h"
#include <stdio.h>
#include "print-stats.h" //para imprimir as estatisticas em rime
#include "rimestats.h"


static void recebido(struct abc_conn *c)
{
  printf("Um pacote chegou no canal, a mensagem eh: '%s'\n", (char *)packetbuf_dataptr());
}


static struct abc_conn conexao_abc;
static const struct abc_callbacks abc_call = {recebido};// declarando a função de call-back

PROCESS(abc_process, "Módulo abc");
AUTOSTART_PROCESSES(&abc_process);
```

```
PROCESS_THREAD(abc_process, ev, data)
{
  PROCESS_BEGIN();

        static struct etimer et;
        abc_open(&conexao_abc, 130, &abc_call); // deve-se escolher um número maior ou igual a 128

        while(1) {
                etimer_set(&et, CLOCK_SECOND);

                PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

                packetbuf_copyfrom("OI", 3);
                abc_send(&conexao_abc);
                printf("Mensagem enviada\n");
                print_stats();
           }

  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

ANEXO I – *Makefile* da simulação do TCP

```
CONTIKI=/home/user/contiki/
WITH_UIP=1
UIP_CONF_IPV6=1
UIP_STATISTICS=1
UIP_CONF_IPV6_CHECKS=1
include $(CONTIKI)/Makefile.include
```

ANEXO J – Código do cliente da simulação do TCP

```
/*
 * \file
 *         TCP project - Client
 * \author
 *
 *           Diego Assis Siqueira Gois - diego.se.ita@gmail.com
 *           João Paulo Andrade Lima - dm.joaopaulo@gmail.com
 *
 */
#include"contiki.h"
#include"contiki-net.h"
#include<stdio.h>
#include "cc2420.h"
#define UIP_APPCALL example1_app

static struct uip_conn *c;
static struct etimer et;
extern struct uip_stats uip_stat;
static int rssi;
static char rssi2;
void example1_connect(void){

        uip_ip6addr_t ip6addr;
```

```
	uip_ip6addr(&ip6addr, 0xfe80, 0x0000, 0x0000, 0x0000, 0x0212, 0x7402, 0x0002,
0x0202);
	c = tcp_connect(&ip6addr, uip_htons(1234), NULL);
	printf("Tentando conectar\n");
}



void enviar_welcome(void){
   printf("Cliente enviando welcome\n");
   uip_send("Welcome!\n", 9);
   printf("Mensagem welcome enviada pelo cliente\n");
   }


PROCESS(cliente_process, "Processo cliente");
AUTOSTART_PROCESSES(&cliente_process);

PROCESS_THREAD(cliente_process, ev, data){

	PROCESS_BEGIN();

	etimer_set(&et,CLOCK_SECOND);
	PROCESS_WAIT_EVENT_UNTIL(timer_expired(&et));
	etimer_reset(&et);
	uip_init();
		//Tentando conectar
	example1_connect();
	while(1){
		PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
		enviar_welcome();
		//Mensagem welcome enviada pelo cliente
		PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
		//example1_app();//é para funcionar sem chamar, não funcionar sem chamar
```

```
                printf ("Pacotes ip perdidos: %d\n",uip_stat.ip.drop);
                printf ("Pacotes ip recebidos: %d\n",uip_stat.ip.recv);
                printf ("Pacotes ip enviados: %d\n",uip_stat.ip.sent);
                printf ("Pacotes ip perdidos versao: %d\n",uip_stat.ip.vhlerr);
                printf ("Pacotes ip perdidos comp ip: %d\n",uip_stat.ip.hblenerr);
                printf ("Pacotes ip perdidos low byte: %d\n",uip_stat.ip.lblenerr);
                printf ("Pacotes ip perdidos fragmentos: %d\n",uip_stat.ip.fragerr);
                printf ("Pacotes ip perdidos err checksum: %d\n",uip_stat.ip.chkerr);
                printf ("Pacotes ip perdidos não eh nada: %d\n",uip_stat.ip.protoerr);

                //Obs: pegando estatísticas tcp
                printf ("Pacotes tcp perdidos: %d\n",uip_stat.tcp.drop);
                printf ("Pacotes tcp recebidos: %d\n",uip_stat.tcp.recv);
                printf ("Pacotes tcp enviados: %d\n",uip_stat.tcp.sent);
                printf ("Pacotes tcp perdidos erro checksum: %d\n",uip_stat.tcp.chkerr);
                printf ("Pacotes tcp perdidos erro de ACK: %d\n",uip_stat.tcp.ackerr);
                printf ("Pacotes tcp segmentos RST reset segments: %d\n",uip_stat.tcp.rst);
                printf ("Pacotes tcp segmentos retransmitidos: %d\n",uip_stat.tcp.rexmit);
                printf ("Pacotes tcp SYNs perdidos por poucas conexões disponíveis: %d\n",uip_stat.tcp.syndrop);
                printf ("Numero de pacotes SYN por porta fechada disparando um RST: %d\n",uip_stat.tcp.synrst);
                //Obs: EStudo comprovou que o rssi não pode ser usado para avaliar a distância do nós
                rssi = cc2420_rssi(); // Leitura de piso RSSI
                rssi2 = cc2420_last_rssi; // RSSI do último pacote recebido
                printf("Leitura de piso RSSI: %d dBm\n",rssi-45); // Esse 45 é o valor de offset
                printf("RSSI do último pacote recebido: %d dBm\n",rssi2-45);

 }
   PROCESS_END();
}
```

ANEXO K – Código do servidor da simulação do TCP

```
/*
 * \file
 *         TCP project - Server
 * \author
 *
 *           Diego Assis Siqueira Gois - diego.se.ita@gmail.com
 *           João Paulo Andrade Lima - dm.joaopaulo@gmail.com
 *
 */
#include<stdio.h>
#include"contiki.h"
#include"contiki-net.h"
#include "cc2420.h"
#define UIP_APPCALL       example2_app
#define UIP_APPSTATE_SIZE sizeof(struct example2_state)


extern struct uip_stats uip_stat;
static int rssi;
static char rssi2;
void example2_connect(void){

   tcp_listen(UIP_HTONS(1234));
   printf("Escutando\n");
}


void example2_app(void) {
          if(uip_newdata() || uip_rexmit()){
          printf("Chegou Dados no servidor\n");
          }
}
```

```
PROCESS(exemplo_2_process, "exemplo 2 servidor");
AUTOSTART_PROCESSES(&exemplo_2_process);
PROCESS_THREAD(exemplo_2_process, ev, data){

        PROCESS_BEGIN();

        uip_init();
        printf("Iniciando a escuta\n");
        example2_connect();

        while(1){
        PROCESS_WAIT_EVENT_UNTIL(ev==tcpip_event);
        example2_app();
                printf ("Pacotes ip perdidos: %d\n",uip_stat.ip.drop);
                printf ("Pacotes ip recebidos: %d\n",uip_stat.ip.recv);
                printf ("Pacotes ip enviados: %d\n",uip_stat.ip.sent);
                printf ("Pacotes ip perdidos versao: %d\n",uip_stat.ip.vhlerr);
                printf ("Pacotes ip perdidos comp ip: %d\n",uip_stat.ip.hblenerr);
                printf ("Pacotes ip perdidos low byte: %d\n",uip_stat.ip.lblenerr);
                printf ("Pacotes ip perdidos fragmentos: %d\n",uip_stat.ip.fragerr);
                printf ("Pacotes ip perdidos err checksum: %d\n",uip_stat.ip.chkerr);
                printf ("Pacotes ip perdidos não eh nada: %d\n",uip_stat.ip.protoerr);

                //Obs: pegando estatísticas tcp
                printf ("Pacotes tcp perdidos: %d\n",uip_stat.tcp.drop);
                printf ("Pacotes tcp recebidos: %d\n",uip_stat.tcp.recv);
                printf ("Pacotes tcp enviados: %d\n",uip_stat.tcp.sent);
                printf ("Pacotes tcp perdidos erro checksum: %d\n",uip_stat.tcp.chkerr);
                printf ("Pacotes tcp perdidos erro de ACK: %d\n",uip_stat.tcp.ackerr);
                printf ("Pacotes tcp segmentos RST reset segments: %d\n",uip_stat.tcp.rst);
                printf ("Pacotes tcp segmentos retransmitidos: %d\n",uip_stat.tcp.rexmit);
                printf ("Pacotes tcp SYNs perdidos por poucas conexões disponíveis:
%d\n",uip_stat.tcp.syndrop);
```

```
        printf ("Numero de pacotes SYN por porta fechada disparando um RST: %d\n",uip_stat.tcp.synrst);
        //Obs: EStudo comprovou que o rssi não pode ser usado para avaliar a distância do nós
        rssi = cc2420_rssi(); // Leitura de piso RSSI
        rssi2 = cc2420_last_rssi; // RSSI do último pacote recebido
        printf("Leitura de piso RSSI: %d dBm\n",rssi-45); // Esse 45 é o valor de offset
        printf("RSSI do último pacote recebido: %d dBm\n",rssi2-45);

    }
    PROCESS_END();
    }
```

ANEXO L – *Makefile* da simulação cliente – servidor REST

```
all: rest-server-example coap-client-example
ifndef TARGET
TARGET=sky
endif

CONTIKI=/home/user/contiki

WITH_UIP6=1
UIP_CONF_IPV6=1
UIP_STATISTICS=1

WITH_COAP = 1

CFLAGS += -DPROJECT_CONF_H=\"project-conf.h\"

ifeq ($(WITH_COAP), 1)
CFLAGS += -DWITH_COAP
APPS += rest-coap
```

```
else
CFLAGS += -DWITH_HTTP
APPS += rest-http
endif

include $(CONTIKI)/Makefile.include

$(CONTIKI)/tools/tunslip6:  $(CONTIKI)/tools/tunslip6.c
	(cd $(CONTIKI)/tools && $(MAKE) tunslip6)

connect-router:	$(CONTIKI)/tools/tunslip6
	sudo $(CONTIKI)/tools/tunslip6 aaaa::1/64

connect-router-cooja: $(CONTIKI)/tools/tunslip6
	sudo $(CONTIKI)/tools/tunslip6 -a 127.0.0.1 aaaa::1/64
```

ANEXO M – Arquivo “project-conf.h” da simulação cliente – servidor REST

```
/*
 * Copyright (c) 2010, Swedish Institute of Computer Science.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the Institute nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
```

```
 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *
 */

#ifndef __PROJECT_RPL_WEB_CONF_H__
#define __PROJECT_RPL_WEB_CONF_H__

#ifndef QUEUEBUF_CONF_NUM
#define QUEUEBUF_CONF_NUM        6
#endif

#ifndef UIP_CONF_BUFFER_SIZE
#define UIP_CONF_BUFFER_SIZE    140
```

```
#endif

#ifndef UIP_CONF_RECEIVE_WINDOW
#define UIP_CONF_RECEIVE_WINDOW  60
#endif

#ifndef WEBSERVER_CONF_CFS_CONNS
#define WEBSERVER_CONF_CFS_CONNS 2
#endif

#endif /* __PROJECT_RPL_WEB_CONF_H__ */
```

ANEXO N – Código do cliente da simulação cliente – servidor REST

```
/*
 *
 *  João Paulo Andrade Lima - dm.joaopaulo@gmail.com
 *  Diego Assis Siqueira Gois - diego.se.ita@gmail.com
 *
 * Based on
 * \file
 *     REST example
 * \author
 *      Adams Dunkels
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "contiki.h"
#include "contiki-net.h"
#include "rest.h"
#include "buffer.h"

#include "cc2420.h" //USAR RSSI
```

```
#define DEBUG 1
#if DEBUG
#include <stdio.h>
#define PRINTF(...) printf(__VA_ARGS__)
#define PRINT6ADDR(addr) PRINTF("%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x", ((uint8_t *)addr)[0], ((uint8_t *)addr)[1], ((uint8_t *)addr)[2], ((uint8_t *)addr)[3], ((uint8_t *)addr)[4], ((uint8_t *)addr)[5], ((uint8_t *)addr)[6], ((uint8_t *)addr)[7], ((uint8_t *)addr)[8], ((uint8_t *)addr)[9], ((uint8_t *)addr)[10], ((uint8_t *)addr)[11], ((uint8_t *)addr)[12], ((uint8_t *)addr)[13], ((uint8_t *)addr)[14], ((uint8_t *)addr)[15])
#define PRINTLLADDR(lladdr) PRINTF(" %02x:%02x:%02x:%02x:%02x:%02x ",(lladdr)->addr[0], (lladdr)->addr[1], (lladdr)->addr[2], (lladdr)->addr[3],(lladdr)->addr[4], (lladdr)->addr[5])
#else
#define PRINTF(...)
#define PRINT6ADDR(addr)
#define PRINTLLADDR(addr)
#endif

#define SERVER_NODE(ipaddr)   uip_ip6addr(ipaddr, 0xfe80, 0, 0, 0, 0x0212, 0x7401, 0x0001, 0x0101)
#define LOCAL_PORT 61617
#define REMOTE_PORT 61616

char temp[100];
int xact_id;
static uip_ipaddr_t server_ipaddr;
static struct uip_udp_conn *client_conn;
static struct etimer et;
#define MAX_PAYLOAD_LEN   100

#define NUMBER_OF_URLS 3
char* service_urls[NUMBER_OF_URLS] = {"light", ".well-known/core", "helloworld"};
```

```
int rssi;
char rssi2;


static void
response_handler(coap_packet_t* response)
{
  uint16_t payload_len = 0;
  uint8_t* payload = NULL;
  payload_len = coap_get_payload(response, &payload);

  PRINTF("Response transaction id: %u", response->tid);
  if (payload) {
    memcpy(temp, payload, payload_len);
    temp[payload_len] = 0;
    PRINTF(" payload: %s\n", temp);
  }
}

static void
send_data(void)
{
  char buf[MAX_PAYLOAD_LEN];

  if (init_buffer(COAP_DATA_BUFF_SIZE)) {
    int data_size = 0;
    int service_id = random_rand() % NUMBER_OF_URLS;
    coap_packet_t* request = (coap_packet_t*)allocate_buffer(sizeof(coap_packet_t));
    init_packet(request);

    coap_set_method(request, COAP_GET);
    request->tid = xact_id++;
    request->type = MESSAGE_TYPE_CON;
    coap_set_header_uri(request, service_urls[service_id]);
```

```
    data_size = serialize_packet(request, buf);

    PRINTF("Client sending request to:[");
    PRINT6ADDR(&client_conn->ripaddr);
    PRINTF("]:%u/%s\n", (uint16_t)REMOTE_PORT, service_urls[service_id]);
    uip_udp_packet_send(client_conn, buf, data_size);

    delete_buffer();
  }
}

static void
handle_incoming_data()
{
  PRINTF("Incoming packet size: %u \n", (uint16_t)uip_datalen());
  if (init_buffer(COAP_DATA_BUFF_SIZE)) {
    if (uip_newdata()) {
      coap_packet_t* response = (coap_packet_t*)allocate_buffer(sizeof(coap_packet_t));
      if (response) {
        parse_message(response, uip_appdata, uip_datalen());
        response_handler(response);
      }
    }
    delete_buffer();
  }
}

PROCESS(coap_client_example, "COAP Client Example");
AUTOSTART_PROCESSES(&coap_client_example);

PROCESS_THREAD(coap_client_example, ev, data)
{
  PROCESS_BEGIN();
```

```
SERVER_NODE(&server_ipaddr);

/* new connection with server */
client_conn = udp_new(&server_ipaddr, UIP_HTONS(REMOTE_PORT), NULL);
udp_bind(client_conn, UIP_HTONS(LOCAL_PORT));

PRINTF("Created a connection with the server ");
PRINT6ADDR(&client_conn->ripaddr);
PRINTF(" local/remote port %u/%u\n",
UIP_HTONS(client_conn->lport), UIP_HTONS(client_conn->rport));

etimer_set(&et, 5 * CLOCK_SECOND);
while(1) {
  PROCESS_YIELD();
  if (etimer_expired(&et)) {
    send_data();
    etimer_reset(&et);

        printf ("Pacotes ip perdidos: %d\n",uip_stat.ip.drop);
        printf ("Pacotes ip recebidos: %d\n",uip_stat.ip.recv);
        printf ("Pacotes ip enviados: %d\n",uip_stat.ip.sent);
        printf ("Pacotes ip perdidos versao: %d\n",uip_stat.ip.vhlerr);
        printf ("Pacotes ip perdidos high byte: %d\n",uip_stat.ip.hblenerr);
        printf ("Pacotes ip perdidos low byte: %d\n",uip_stat.ip.lblenerr);
        printf ("Pacotes ip perdidos fragmentos: %d\n",uip_stat.ip.fragerr);
        printf ("Pacotes ip perdidos err checksum: %d\n",uip_stat.ip.chkerr);
        printf ("Pacotes ip perdidos que não eh TCP, UDP, ICMP: %d\n",uip_stat.ip.protoerr);

        printf ("Pacotes udp perdidos: %d\n",uip_stat.udp.drop);
        printf ("Pacotes udp recebidos: %d\n",uip_stat.udp.recv);
        printf ("Pacotes udp enviados: %d\n",uip_stat.udp.sent);
        printf ("Pacotes udp perdidos err checksum: %d\n",uip_stat.udp.chkerr);

        rssi = cc2420_rssi(); // Leitura de piso RSSI
```

```
        rssi2 = cc2420_last_rssi; // RSSI do último pacote recebido
        printf("Leitura de piso RSSI: %d dBm\n",rssi-45); // Esse 45 é o valor de offset
        printf("RSSI do último pacote recebido: %d dBm\n",rssi2-45);
    } else if (ev == tcpip_event) {
      handle_incoming_data();

        printf ("Pacotes ip perdidos: %d\n",uip_stat.ip.drop);
        printf ("Pacotes ip recebidos: %d\n",uip_stat.ip.recv);
        printf ("Pacotes ip enviados: %d\n",uip_stat.ip.sent);
        printf ("Pacotes ip perdidos versao: %d\n",uip_stat.ip.vhlerr);
        printf ("Pacotes ip perdidos high byte: %d\n",uip_stat.ip.hblenerr);
        printf ("Pacotes ip perdidos low byte: %d\n",uip_stat.ip.lblenerr);
        printf ("Pacotes ip perdidos fragmentos: %d\n",uip_stat.ip.fragerr);
        printf ("Pacotes ip perdidos err checksum: %d\n",uip_stat.ip.chkerr);
        printf ("Pacotes ip perdidos que não eh TCP, UDP, ICMP: %d\n",uip_stat.ip.protoerr);

        printf ("Pacotes udp perdidos: %d\n",uip_stat.udp.drop);
        printf ("Pacotes udp recebidos: %d\n",uip_stat.udp.recv);
        printf ("Pacotes udp enviados: %d\n",uip_stat.udp.sent);
        printf ("Pacotes udp perdidos err checksum: %d\n",uip_stat.udp.chkerr);

        rssi = cc2420_rssi(); // Leitura de piso RSSI
        rssi2 = cc2420_last_rssi; // RSSI do último pacote recebido
        printf("Leitura de piso RSSI: %d dBm\n",rssi-45); // Esse 45 é o valor de offset
        printf("RSSI do último pacote recebido: %d dBm\n",rssi2-45);
    }

  }

  PROCESS_END();
}
```

ANEXO O – Código do servidor da simulação cliente – servidor REST

```
/*
 * \file
 *     REST example
 * \author
 *      Adams Dunkels
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "contiki.h"
#include "contiki-net.h"
#include "rest.h"

#if defined (PLATFORM_HAS_LIGHT)
#include "dev/light-sensor.h"
#endif
#if defined (PLATFORM_HAS_BATT)
#include "dev/battery-sensor.h"
#endif
#if defined (PLATFORM_HAS_SHT11)
#include "dev/sht11-sensor.h"
#endif
#if defined (PLATFORM_HAS_LEDS)
#include "dev/leds.h"
#endif

#define DEBUG 1
#if DEBUG
#include <stdio.h>
#define PRINTF(...) printf(__VA_ARGS__)
#define                    PRINT6ADDR(addr)                    PRINTF("
%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%
```

```
02x ", ((uint8_t *)addr)[0], ((uint8_t *)addr)[1], ((uint8_t *)addr)[2], ((uint8_t *)addr)[3], ((uint8_t *)addr)[4], ((uint8_t *)addr)[5], ((uint8_t *)addr)[6], ((uint8_t *)addr)[7], ((uint8_t *)addr)[8], ((uint8_t *)addr)[9], ((uint8_t *)addr)[10], ((uint8_t *)addr)[11], ((uint8_t *)addr)[12], ((uint8_t *)addr)[13], ((uint8_t *)addr)[14], ((uint8_t *)addr)[15])
#define PRINTLLADDR(lladdr) PRINTF(" %02x:%02x:%02x:%02x:%02x:%02x ",(lladdr)->addr[0], (lladdr)->addr[1], (lladdr)->addr[2], (lladdr)->addr[3],(lladdr)->addr[4], (lladdr)->addr[5])
#else
#define PRINTF(...)
#define PRINT6ADDR(addr)
#define PRINTLLADDR(addr)
#endif

char temp[100];

/* Resources are defined by RESOURCE macro, signature: resource name, the http methods it handles and its url*/
RESOURCE(helloworld, METHOD_GET, "helloworld");

/* For each resource defined, there corresponds an handler method which should be defined too.
 * Name of the handler method should be [resource name]_handler
 * */
void
helloworld_handler(REQUEST* request, RESPONSE* response)
{
  sprintf(temp,"Hello World!\n");

  rest_set_header_content_type(response, TEXT_PLAIN);
  rest_set_response_payload(response, (uint8_t*)temp, strlen(temp));
}

RESOURCE(discover, METHOD_GET, ".well-known/core");
void
```

```
discover_handler(REQUEST* request, RESPONSE* response)
{
  char temp[100];
  int index = 0;
  index += sprintf(temp + index, "%s,", "</helloworld>;n=\"HelloWorld\"");
#if defined (PLATFORM_HAS_LEDS)
  index += sprintf(temp + index, "%s,", "</led>;n=\"LedControl\"");
#endif
#if defined (PLATFORM_HAS_LIGHT)
  index += sprintf(temp + index, "%s", "</light>;n=\"Light\"");
#endif

  rest_set_response_payload(response, (uint8_t*)temp, strlen(temp));
  rest_set_header_content_type(response, APPLICATION_LINK_FORMAT);
}

#if defined (PLATFORM_HAS_LIGHT)
uint16_t light_photosynthetic;
uint16_t light_solar;

void
read_light_sensor(uint16_t* light_1, uint16_t* light_2)
{
  *light_1 = light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHETIC);
  *light_2 = light_sensor.value(LIGHT_SENSOR_TOTAL_SOLAR);
}

/*A simple getter example. Returns the reading from light sensor with a simple etag*/
RESOURCE(light, METHOD_GET, "light");
void
light_handler(REQUEST* request, RESPONSE* response)
{
  read_light_sensor(&light_photosynthetic, &light_solar);
  sprintf(temp,"%u;%u", light_photosynthetic, light_solar);
```

```
  char etag[4] = "ABCD";
  rest_set_header_content_type(response, TEXT_PLAIN);
  rest_set_header_etag(response, etag, sizeof(etag));
  rest_set_response_payload(response, temp, strlen(temp));
}
#endif /*PLATFORM_HAS_LIGHT*/

#if defined (PLATFORM_HAS_LEDS)
/*A simple actuator example, depending on the color query parameter and post variable mode,
corresponding led is activated or deactivated*/
RESOURCE(led, METHOD_POST | METHOD_PUT , "led");

void
led_handler(REQUEST* request, RESPONSE* response)
{
  char color[10];
  char mode[10];
  uint8_t led = 0;
  int success = 1;

  if (rest_get_query_variable(request, "color", color, 10)) {
    PRINTF("color %s\n", color);

    if (!strcmp(color,"red")) {
      led = LEDS_RED;
    } else if(!strcmp(color,"green")) {
      led = LEDS_GREEN;
    } else if ( !strcmp(color,"blue") ) {
      led = LEDS_BLUE;
    } else {
      success = 0;
    }
  } else {
    success = 0;
```

```
  }

  if (success && rest_get_post_variable(request, "mode", mode, 10)) {
    PRINTF("mode %s\n", mode);

    if (!strcmp(mode, "on")) {
      leds_on(led);
    } else if (!strcmp(mode, "off")) {
      leds_off(led);
    } else {
      success = 0;
    }
  } else {
    success = 0;
  }

  if (!success) {
    rest_set_response_status(response, BAD_REQUEST_400);
  }
}


/*A simple actuator example. Toggles the red led*/
RESOURCE(toggle, METHOD_GET | METHOD_PUT | METHOD_POST, "toggle");
void
toggle_handler(REQUEST* request, RESPONSE* response)
{
  leds_toggle(LEDS_RED);
}
#endif /*defined (CONTIKI_HAS_LEDS)*/


PROCESS(rest_server_example, "Rest Server Example");
AUTOSTART_PROCESSES(&rest_server_example);
```

```
PROCESS_THREAD(rest_server_example, ev, data)
{
  PROCESS_BEGIN();

#ifdef WITH_COAP
  PRINTF("COAP Server\n");
#else
  PRINTF("HTTP Server\n");
#endif

  rest_init();

#if defined (PLATFORM_HAS_LIGHT)
  SENSORS_ACTIVATE(light_sensor);
  rest_activate_resource(&resource_light);
#endif
#if defined (PLATFORM_HAS_LEDS)
  rest_activate_resource(&resource_led);
  rest_activate_resource(&resource_toggle);
#endif /*defined (PLATFORM_HAS_LEDS)*/

  rest_activate_resource(&resource_helloworld);
  rest_activate_resource(&resource_discover);

  PROCESS_END();
}
```